Trabajo Fin de Grado

OPTIMIZACIÓN DEL FLUJO DE PROGRAMA EN SOFTWARE DE GESTIÓN PARA MICROELECTRÓNICA

Autor: Pedro Manuel Moreno Marcos
Tutor: Víctor Pedro Gil Jiménez
Director: Álvaro Padierna Díaz

Universidad Carlos III de Madrid
Escuela Politécnica Superior

Trabajo Fin de Grado

Optimización del flujo de programa en software de gestión para microelectrónica

Autor

Pedro Manuel Moreno Marcos

Tutor

Víctor Pedro Gil Jiménez

Director

Álvaro Padierna Díaz

EL TRIBUNAL

Presidente: Eva Rajo Iglesias

Secretario: Marta Portela García

Vocal: Alicia Rodríguez Carrión

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día 8 de julio de 2015 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, el tribunal acuerda otorgarle la calificación de

PRESIDENTE VOCAL SECRETARIO

Agradecimientos

La realización de este Trabajo Fin de Grado ha sido posible gracias a la ayuda, de forma directa o indirecta, de muchas personas a las que agradezco por su apoyo y dedicación.

Le agradezco a la empresa (Crisa) por haberme dado la oportunidad de conseguir mi primera experiencia laboral y brindarme la oportunidad de la realización del Trabajo Fin de Grado con ellos.

Quiero dar las gracias a Álvaro Padierna, mi tutor en la empresa, por su apoyo, dedicación y paciencia, que ha sido esencial para la consecución de los objetivos y el aprendizaje.

No quiero olvidarme de Juan Antonio Ortega, el responsable de microelectrónica, por su dedicación y supervisión en el proyecto.

Mención especial a Víctor P. Gil, mi tutor académico, que desde el primer momento confió en mí para realizar el proyecto, por su colaboración, interés y consejos para la elaboración de este documento.

Debo dar también gracias a mis padres, mi abuelo, mi tía y el resto de mi familia por su apoyo y cariño durante toda esta larga travesía.

Por último, también es necesario mencionar a todo el profesorado que he tenido porque ellos han sido el medio para sentar las bases para poder llegar hasta aquí y a aquellos compañeros a los que también hay que agradecer por su colaboración. Muchas gracias a todos.

Resumen

La empresa Computadoras Redes e Ingeniería, S.A. (*Crisa*), dentro de su sección de microelectrónica necesita una herramienta que se encargue de instalar paquetes, que incluyen diseños de *hardware* que pueden ser reutilizados en varios proyectos. El propósito de este Trabajo Fin de Grado es el de implementar la aplicación a medida que satisfaga las necesidades planteadas.

Para ello, se pretende analizar las características que ofrecen los sistemas de gestión de paquetes actuales, dado que el software especificado se puede considerar como tal, para poder extraer las características que pueden ser reaprovechadas y conocer cuáles son las estructuras de datos que permiten el funcionamiento de estas herramientas.

El desarrollo se separa en dos versiones diferenciadas y consistentes, pero que utilizan un núcleo común. Por una parte, se encuentra la versión en línea de comandos y por otra una versión gráfica. Ambas proporcionan la misma funcionalidad pero los distintos usuarios que prueban el programa tienden más al uso de una u otra y por ello, debe garantizarse que no haya diferencia entre usar una u otra. Del mismo modo, un aspecto importante es el de la compatibilidad, puesto que no todos los desarrolladores que utilizan la herramienta utilizan la misma plataforma y el objetivo es que sea accesible a todos.

Palabras clave: paquete, dependencia, *Git*, instalación.

Abstract

Computadoras, Redes e Ingeniería, S.A. (Crisa) needs a tool for its microelectronic section which handles the problem of installing packages, which include hardware designs that can be reused in other projects. The purpose of this Bachelor Thesis is to implement a custom-made application which satisfies the exposed necessities.

To do that, it is intended to analyze the features other package manager systems offer, as this software can be considered as such, to be able to extract the reusable features and to know what the common data structures, which allows those tools working, are.

In order to develop the tool, work has been separated in two consistent and differenced versions, which use the same kernel. On the one hand, there is a command-line version and on the other hand there is a graphic one. Both provide the same functionality but users tend to user one more than another and there must be guaranteed that there is no difference between using any of those versions. At the same time, an important aspect is the compatibility between platforms since each developer which uses the tool can use a different one and the main objective is that all of them can use the program independently on where they run it.

Key words: package, dependence, *Git*, installation.

Índice general

1. Introducción **1**

1.1. English version 1

1.1.1. Motivation 1

1.1.2. Objectives and needs 1

1.1.3. Document structure 3

1.2. Versión en castellano 4

1.2.1. Motivación 4

1.2.2. Necesidades y objetivos 4

1.2.3. Estructura de la memoria 6

2. Estado del arte **8**

2.1. *APT: Advanced Package Tool* 10

2.1.1. Fichero de fuentes (*Sources list*) 10

2.1.2. APT-GET 11

2.1.3. Verificación de paquetes 12

2.1.4. Actualización de paquetes 13

2.1.5. Interfaces: *Aptitude*, *Synaptic* 13

2.2. *YUM*: *Yellow Dog Updater Modified* 14

2.2.1. Opciones principales de *YUM* 16

2.2.2. Conclusiones 17

2.3. *OneGet* 17

3. Definición de los metadatos **20**

3.1. Formato de fuentes 20

3.2. Nomenclatura de versiones 24

3.2.1. Formato de tags 25

3.2.2. Definición de los metadatos del paquete 25
3.3. Definición del fichero de instalación 29
3.4. Datos sobre la instalación 31

4. Primeras funcionalidades del programa 35
4.1. Ayuda del programa 35
4.2. Versión 37
4.3. Listado de paquetes 37
4.4. Selección del fichero de fuentes 40
4.5. Selección del fichero de instalación 41

5. Instalación de paquetes 42
5.1. Comprobación del estado del repositorio 42
5.1.1. Definición de la raíz del proyecto 44
5.2. Obtención de la lista de paquetes disponibles 45
5.3. Obtención de la lista de paquetes requeridos 46
5.4. Resolución de dependencias 46
5.4.1. Fichero de conflictos 54
5.5. Preparación de la lista de instalación final I: Análisis de los paquetes instalados. 56
5.6. Preparación de la lista de instalación final II: Comprobación de la integridad . . 57
5.7. Desinstalación de paquetes 59
5.7.1. Sistema de *backup* durante la instalación 59
5.8. Copia de ficheros 60
5.9. Establecimiento de permisos 62
5.10. Ejecución de los comandos de post-instalación 63
5.11. Actualización del fichero de datos de la instalación 64
5.12. Actualización del fichero de instalación 64

5.13. Ejemplo de instalación 65

6. Funcionalidades del programa complementarias **67**

6.1. Comprobación de la integridad de un paquete 67

6.1.1. Ejemplos de comprobación de paquetes 69

6.2. Forzar la instalación 70

6.3. Desinstalación de paquetes 70

6.4. Listar los paquetes instalados 72

6.5. Instalación específica 74

6.6. Modo de instalación no automático 75

6.7. Limpieza del directorio caché 75

6.8. Verbosidad del programa 76

6.8.1. Historial de mensajes (`myum.log`) 77

6.9. Actualización de paquetes 77

6.10. Borrado automático 78

6.11. Historial de instalación 79

6.12. Modo de simulación 79

7. Interfaz gráfica **82**

7.1. Barra de menús 82

7.2. Barra de opciones 84

7.3. Panel principal 85

7.3.1. Sistema de marcado de paquetes 85

7.4. Ventana de paquetes en desarrollo 90

7.5. Panel de mensajes 91

8. Desarrollo complementario del proyecto y resultados **92**

8.1. Pruebas del código y cobertura 92

8.2. Compatibilidad del programa 94
8.3. Documentación 95
8.4. Resultados 95

9. Conclusiones **97**

9.1. English version 97
9.2. Versión en castellano 98

Anexos **100**

A. Planificación del proyecto **100**

A.1. Estructura del desglose del trabajo (*EDT*) 100
A.2. Secuenciación de las tareas 100
A.3. Diagrama de Gantt 104

B. Presupuesto del proyecto **107**

B.1. Recursos materiales 107
B.2. Recursos humanos 108
B.3. Presentación del presupuesto 108

C. Marco regulador **110**

C.1. Marco legal 110
C.2. Marco técnico 110

D. Entorno socio-económico **112**

D.1. Entorno social 112
D.2. Entorno económico 112
D.2.1. Descripción de la empresa y modelo de negocio 112
D.3. Aportación al modelo de negocio 114

E. Summary in English **115**

E.1. Introduction 115

E.2. State of art 115

E.3. Metadata definition 116

E.4. First functionalities of the program 117

E.5. Package installation 117

E.6. Complementary functions of the program 119

E.7. Graphic interface 120

E.8. Complementary project development and results 122

E.9. Conclusions 123

Referencias **124**

Listado de figuras

1. General scheme of how the package manager system work in the context of the section. . . . 2
2. Esquema general del funcionamiento del sistema de gestión de paquetes en el contexto de la sección. . . . 5
3. Ejemplo de árbol de dependencias. . . . 9
4. Esquema de funcionamiento de un gestor de paquetes. Fuente: *Opensuse* [2]. . . . 9
5. Gestor de paquetes *aptitude*. . . . 14
6. Gestor de paquetes *synaptic*. . . . 15
7. Distintas opciones de *OneGet*. Fuente: *How-to Geek* [11]. . . . 18
8. Alternativa 1 de formato de fuentes. . . . 21
9. Alternativa 2 de formato de fuentes. . . . 22
10. Alternativa 3 de formato de fuentes. . . . 23
11. Contenido del fichero `myum_repo_content.json`. . . . 27
12. Contenido del fichero `pkg1.json`. . . . 28
13. Formato del paquete *APB*. . . . 29
14. Formato inicial del fichero `myum_install_def.json`. . . . 30
15. Formato del fichero `.myum.db`. . . . 34
16. Opción de ayuda del programa. . . . 36
17. Esquema del procedimiento de obtención de la lista de paquetes. . . . 38
18. Salida de la función `git ls-remote -tags`. . . . 38
19. Salida por pantalla tras listar la lista de paquetes. . . . 40
20. Ciclo de vida del estado de los ficheros. Fuente: *Git*. [1] 43
21. Listado de paquetes requeridos. . . . 46
22. Resolución inicial de dependencias. . . . 47
23. Árbol de dependencias inicial. . . . 48
24. Árbol de dependencias tras podar B_2.0 y B_2.1. . . . 49

25. Árbol de dependencias tras podar A_1.1. 49
26. Árbol de dependencias convergente. 50
27. Diagrama de flujo de la resolución de dependencias. 51
28. Formato del fichero `myum_conflict.json`. 55
29. Resolución de conflictos de versiones incompatibles por el usuario. 55
30. Proceso de obtención de la lista de paquetes a instalar. 58
31. Instalación de paquetes. 66
32. Instalación de paquetes con resolución no automática de conflictos. 66
33. Proceso de comprobación de un paquete. 68
34. Comprobación de un paquete correcta. 70
35. Comprobación de un paquete en el que hay ficheros modificados, añadidos y eliminados. 71
36. Desinstalación errónea al fallar la comprobación de la integridad. 73
37. Desinstalación correcta utilizando la opción de forzado. 73
38. Listado de paquetes antes y después de una instalación. 74
39. Limpieza del directorio caché. 76
40. Actualización de paquetes errónea por falta de repositorio limpio. 78
41. Actualización de paquetes mediante la opción `-up`. 79
42. Borrado automático de paquetes 80
43. Historial de operaciones 80
44. Modo simulación del *myum*. 81
45. Ayuda previa al arranque de la interfaz gráfica 83
46. Vista inicial de la interfaz gráfica 83
47. Marcado de operaciones en la interfaz gráfica. 86
48. Ventana para añadir dependencias a un paquete mediante la interfaz gráfica. . . 91
49. Estructura de descomposición del trabajo. 101
50. Diagrama de Gantt del proyecto. 106

51. Help option of *myum*. 118

52. Graphic user interface. 121

Listado de tablas

1. Resumen comparativo de los distintos gestores de paquetes. 19

2. Comparativa de los dos formatos de `myum_install_def.json`. 32

3. Atributos de permisos en Linux. 62

4. Descripción de los atributos de permisos en Linux. Fuente: The Linux Command Line [18]. 63

5. Comprobaciones cuyo resultado puede ser omitido mediante la opción de forzado. 71

6. Resumen de tareas, duraciones y precedencias del proyecto. 105

7. Presupuesto del proyecto. 109

1. Introducción

1.1. English version

1.1.1. Motivation

Within the context of a company with work in the area of engineering, there are lots of projects with a particular structure which usually follow certain rules for maintenance purposes. Although each project could seem to be very different to others, in most cases there are elements in common and there is an important advantage if these components can be reused.

Nowadays, package managers allow to easily installing software components so that a list of available packages is displayed and user chooses what packages to install. In order to understand it better, suppose you buy a new personal computer. If you have done it before, you would probably find very tedious to install all software you normally use, but would not it be better if you could indicate all as you want and just press a key in your keyboard? If you had a package manager which had all your programs available, that task could be completely automatic.

The motivation of this Bachelor Thesis is to implement a package manager system which allows reusing blocks of code developed in different projects in the context of a real business which requires, as it will be exposed later, a system like this. The fact of being able to develop an application in a professional context is also a motivating fact for the formation and experience which can be obtained while elaborating this project which allows finishing the degree.

1.1.2. Objectives and needs

Microelectronic section of *Crisa* (Computadoras, Redes e Ingeniería, S.A.) works in the development of *FPGAs* (Field Programmable Gate Array) and *ASICs* (Application-Specific Integrated Circuit) designs. While developing them, there are logic blocks, called *IP* (Intelectual Property) which can be reused in different projects.

This section uses *Git* [1] as a revision control system and repositories are created to store projects. Among those, there are the *IPs* that have been mentioned.

The objective is to design a package manager system, which allows different users to install packages corresponding to different *IPs* to be able to use them in the developing projects which requires them, so that if the package required also needs other packages, those other packages called dependencies are also installed. Figure 1 shows how the program works. In this figure, it is supposed that there are four hosts developing projects A and B; and *IPs Crip1* and *Crip2*. While these developers get advances in their projects, they upload the content to the server. That is why the up arrows appear. If *Crip1* requires *Crip2* for working and project A needs *Crip1*, host 1 will connect *VMGitLab* server (server where repositories with projects information are found) and will download the content of both *IPs*.

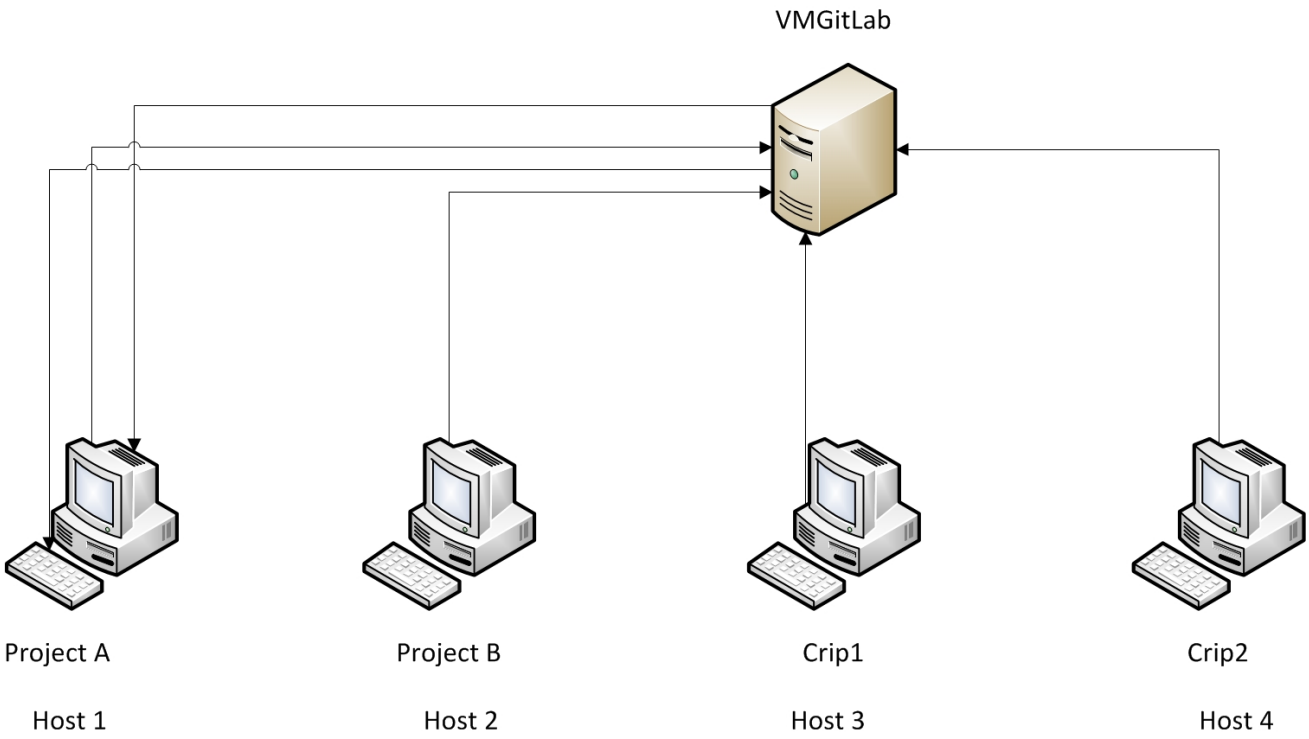

Figura 1: General scheme of how the package manager system work in the context of the section.

In the section, there was a try to develop a program to do that, but it obviated most of the issues that it could carry on and finally it was not used. For this reason, whenever an employee needs a package to develop its project, he must access to *VMGitLab*, looks for the package required and download it manually and place files in the proper place. What is more, if the package depends on another one, he must repeat the same process with the dependence, but if there are nested dependencies, process would be repeated recursively so many times until all the material needed was obtained.

As it has been shown to be very inefficient, the need to design a package manager which automates the whole process describe is clearly justified. What is more, this system needs more features because if the employee which installed the package needs to remove it, he would look for all directories which have files corresponding to the package to delete them. Furthermore there could be cases in which different versions are not compatible and inefficiency could be even higher is after installing a package, it does not work and user has to delete it and install something different. A package manager has enough intelligence to perform all these operations automatically. For this reason, this document explains how to design the package manager which automates processes in the section and raises efficiency. General project's objectives are the followings:

- Design a data structure which allows finding and associating the content of repositories with different packages in their different versions.
- Design the database structure to store the installed packages information and to be able to apply for new packages.

- Implement the package manager which allows to perform operations with the available packages.
- Implement a graphic user interface to make the program easier to use.
- Ensure that the implemented design can be used in every computer in every platform.

1.1.3. Document structure

In order to make this document easier to read and understand and to be able to separate different parts of design, the following chapter's structure has been set:

- Chapter 1: Introduction: Describes needs and motivations to do the project and in its objectives as well as the document structure.
- Chapter 2: State of art: Describes and compares the most important existing package managers.
- Chapter 3: Metadata description: Defines the data structure necessary to locate and get access to packages and to define an installation and register the operations performed.
- Chapter 4: First functionalities of the program: Describes the preliminary functions which are normally used before installing packages, which are mainly the ones to know the available packages and to define the metadata files.
- Chapter 5: Packages installation: Describes the process to install one or more packages defined in the install definition file.
- Chapter 6: Complementary functions of the program: Describes the functionalities the program has apart from the used for the installation process, such as uninstallation, update, or integrity check.
- Chapter 7: Graphic interface: Describes the features and functionalities the graphic interface has.
- Chapter 8: Complementary project development and results: Describes aspects related to the project as the implementation of unit tests which do not give more functionality but they are necessary in the context of the project. Furthermore, there is an analysis of the obtained results.
- Chapter 9: Conclusions: Shows a summary of the content of the content, the difficulties found and the consecution of objectives is analyzed as well as possible complements are shown.

1.2. Versión en castellano

1.2.1. Motivación

Dentro del contexto de una empresa que se dedique a cualquier ámbito de la ingeniería, existe una inmensa cantidad de proyectos con una estructura particular y que normalmente se acoge a ciertos convenios para su buen mantenimiento. Aunque cada proyecto pueda resultar completamente diferente al anterior, en la mayoría de casos pueden existir elementos comunes y el hecho de poder reutilizar estos componentes supone una gran ventaja.

Actualmente, existen los llamados gestores de paquetes que permiten la instalación de forma rápida de componentes de *software* para lo cual se presenta una lista de todos los programas disponibles y se instalan aquellos que el usuario requiera. Para comprender mejor su utilidad, póngase en el caso de que adquiere un nuevo equipo informático. Seguramente que si lo ha hecho con anterioridad le pueda resultar bastante tedioso tener que instalar todo el *software* que normalmente utiliza. y ¿no le sería mucho más ventajoso si pudiera indicar todo lo que se requiere, dar a una tecla y despreocuparse? Si dispusiera de un gestor de paquetes que tuviese disponible todos los programas, esta tarea se automatizaría completamente.

La motivación de este Trabajo Fin de Grado (*TFG*) es la de conseguir implementar un sistema de gestión de paquetes que permite la reutilización de código realizado en distintos proyectos dentro del contexto de una empresa real que requiere, como se expondrá a continuación, de un sistema de estas características. El hecho de poder desarrollar una aplicación con proyección de uso real dentro de un contexto profesional es también un factor motivador para la formación y experiencia que se puede adquirir durante la elaboración del proyecto que permite la finalización del grado.

1.2.2. Necesidades y objetivos

La sección de Microelectrónica de *Crisa* (Computadoras, Redes e Ingeniería, S.A.) trabaja en el desarrollo de diseños propios de *FPGAs* (Field Programmable Gate Array) y *ASICs* (Application-Specific Integrated Circuit). Durante el desarrollo de los mismos, existen bloques de lógica o de datos, llamados *IP* (Intelectual Property), que pueden ser reutilizados en distintos proyectos.

La sección utiliza *Git* [1] como sistema de control de versiones y se crean repositorios donde se almacenan los distintos proyectos. Entre éstos, están los *IPs* que se han comentado anteriormente.

El objetivo es crear un sistema de gestión de paquetes, que permita a los distintos usuarios poder instalar paquetes pertenecientes a los *IPs* para poder utilizarlos durante el desarrollo de un proyecto que los necesite, de modo que si dicho paquete requiere de otros para su funcionamiento, se instalen todas las dependencias necesarias. La figura 2 muestra un claro ejemplo de funcionamiento. En ella se supone que hay cuatro equipos que están desarrollando los proyectos A y B; y los *IPs Crip1* y *Crip2*. Cada uno de los desarrolladores según va avanzando en su proyecto sube el contenido al servidor, de ahí que aparezcan las flechas ascendentes. Si el

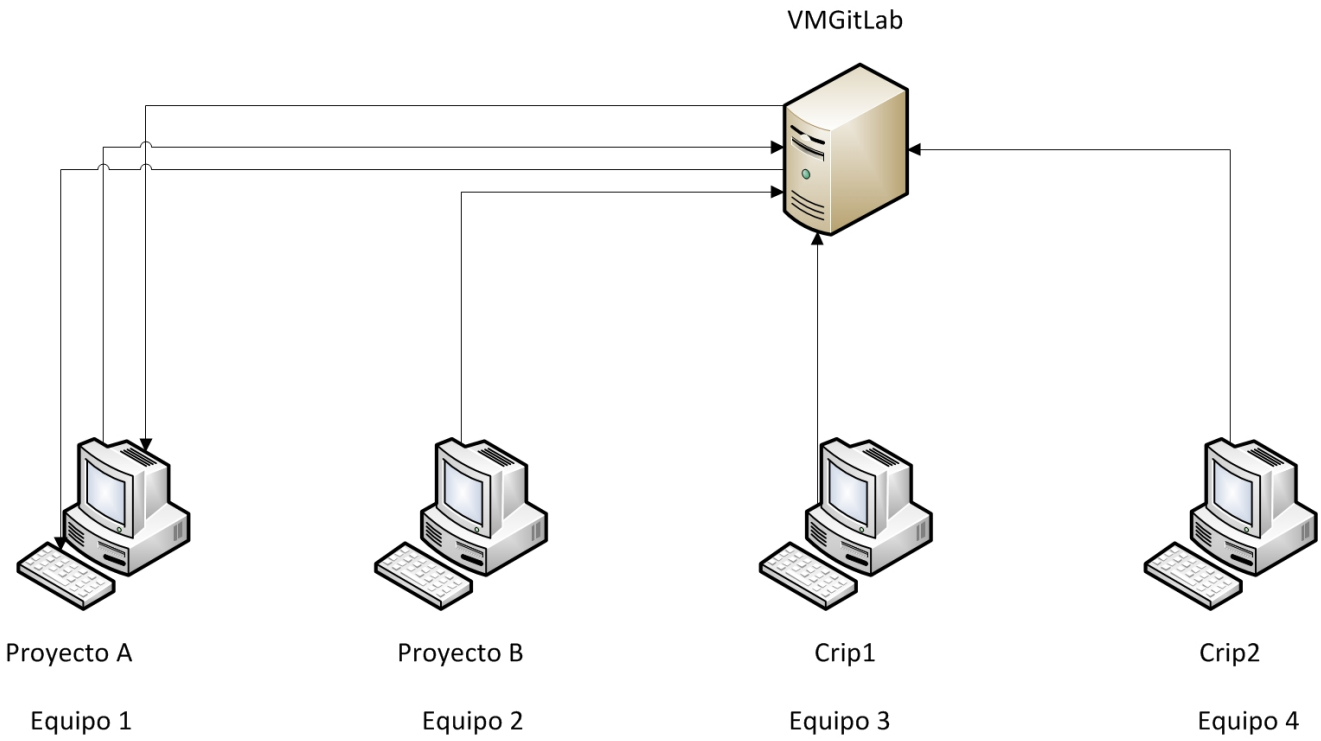

Figura 2: Esquema general del funcionamiento del sistema de gestión de paquetes en el contexto de la sección.

Crip1 requiere para su funcionamiento del *Crip2* y el proyecto A necesita el *Crip1*, entonces el equipo 1 donde se desarrolla el proyecto se conectará al servidor *VMGitLab* (el servidor donde se encuentran los repositorios de los proyectos de la sección) y se descargará el contenido de ambos *IPs*.

En la sección se planteó una versión inicial de este programa, pero que obviaba gran parte de la problemática que conlleva y que al final dio lugar al abandono. Por ello, cuando un empleado necesita un paquete para el desarrollo de su proyecto, debe acceder a *VMGitLab*, buscar el paquete que necesita, descargarlo manualmente y situar los ficheros en el lugar correspondiente. Además, si éste depende de otro, debe hacer lo propio con esta dependencia; y si hay dependencias anidadas, se debería repetir el proceso recursivamente hasta que se tuviera todo el material necesario.

Dado que esto es muy ineficiente, se justifica la necesidad de elaborar un sistema de gestión de paquetes, que automatice todo el proceso descrito anteriormente, y que le dote de una capacidad de acción mayor, pues si más adelante, el empleado que instaló el paquete necesita eliminarlo, debería buscar todos los directorios que tienen archivos de ese paquete para borrarlos. También podría haber casos en los que las distintas versiones de los paquetes no fueran compatibles y la ineficiencia podría ser mucho mayor si después de instalar algo no funciona y hay que borrarlo e instalar algo distinto. Un sistema de gestión de paquetes tiene la inteligencia suficiente para realizar todas estas tareas de forma automática. Por ello, este documento tratará de cómo elaborar el gestor que contribuya a la automatización de los procesos descritos en la sección y el aumento de la eficiencia en la misma. Los objetivos que se plantean desde el inicio son los siguientes:

- Diseñar una estructura de datos que permita localizar y asociar el contenido de los repositorios a los distintos paquetes en sus distintas versiones.
- Diseñar la estructura de base de datos para almacenar el contenido de paquetes instalados y para poder solicitar nuevos paquetes.
- Implementar el sistema de gestión de paquetes que permita realizar las operaciones básicas con los paquetes disponibles.
- Dotar al sistema de una interfaz gráfica de usuario para mejorar su facilidad de uso.
- Asegurar que el diseño implementado pueda ser utilizado desde cualquier equipo independientemente de su plataforma.

1.2.3. Estructura de la memoria

Para facilitar la lectura y poder entender el proyecto y poder diferenciar sus diferentes partes de diseño, se ha estructurado la memoria en diferentes capítulos.

- Capítulo 1: Introducción: Describe las necesidades y motivaciones que llevan a la elaboración del proyecto junto con los objetivos y la estructura de la memoria.
- Capítulo 2: Estado del arte: Describe y compara los principales sistemas de gestión de paquetes existentes.
- Capítulo 3: Descripción de los metadatos: Define la estructura de datos necesaria para la localización y acceso de los paquetes y para definir la instalación y registrar las operaciones realizadas.
- Capítulo 4: Primeras funcionalidades del programa: Describe las funcionalidades preliminares antes de hacer uso de la instalación, que son principalmente las funciones para conocer los paquetes disponibles y definir los ficheros de metadatos.
- Capítulo 5: Instalación de paquetes: Describe el proceso para conseguir instalar uno o varios paquetes definidos en el fichero de definición de la instalación.
- Capítulo 6: Funcionalidades del programa complementarias: Describe las funcionalidades que posee el programa a parte de la instalación, como pueden ser la desinstalación, actualización o comprobación de la integridad, entre otras.
- Capítulo 7: Interfaz gráfica: Describe las características y funcionalidades que posee la interfaz gráfica del programa.
- Capítulo 8: Desarrollo complementario del proyecto y resultados: Describe aspectos relacionados con el desarrollo del proyecto como la implementación de las pruebas unitarias que no dotan al programa de características nuevas, pero son necesarios en el contexto del proyecto. También se realiza un análisis de los resultados obtenidos.

- Capítulo 9: Conclusiones: Presenta una relación de los contenidos presentados, las dificultades encontradas y se analiza la conclusión de los objetivos, así como se presentan posibles complementos al proyecto.

2. Estado del arte

En primer lugar, es necesario adquirir conocimientos sobre la gestión de paquetes y el funcionamiento de los principales gestores. Por ello, en este capítulo se presentarán las nociones básicas de los gestores de paquetes y se expondrá en líneas generales el comportamiento de los dos gestores de paquetes más importantes: *apt-get* y *yum*. Finalmente, se comentará el caso de la entrada del nuevo gestor de paquetes de *Microsoft*.

Sistemas de gestión de paquetes

Se conoce como sistema de gestión de paquetes al conjunto de herramientas que automatizan los procesos de instalación, actualización y eliminación de software. Dicho *software* se encuentra en repositorios, ya sean locales o remotos, y a los cuales se conectan los gestores para, en primer lugar, conseguir los metadatos que contienen la información sobre el nombre, la versión del paquete y en especial las dependencias (aquellos paquetes necesarios para que puada funcionar el *software* que se desea instalar). Con toda la información, se procede a construir un árbol de dependencias con todos los paquetes necesarios y se intenta evitar conflictos de versiones. Una vez realizado esto, se procede a descargar el contenido de los paquetes de los repositorios y se efectúa la instalación. Tras la misma, se almacenan metadatos en una base de datos local para poder llevar cuenta de los paquetes y las operaciones realizadas. Los elementos principales que intervienen en un sistema de gestión de paquetes son:

- Paquetes: Son el conjunto de ficheros que componen un software (puede ser una aplicación en sí misma, o paquetes de desarrollo que sirvan para el diseño de otros componentes...). Es lo que el usuario solicita (directa o indirectamente) para realizar la instalación. El volumen de contenido depende de la aplicación específica.
- Dependencias: En muchas ocasiones, para poder ejecutar una aplicación se requieren herramientas, bibliotecas, etc. Este contenido, imprescindible para poder ejecutar el contenido del paquete principal, son otros paquetes que se deben instalar junto con el paquete principal y al depender éste de dichos paquetes, se les denomina dependencias. Es posible incluso que lo que para una instalación sea un paquete principal para otra sea una dependencia. Suponga que un usuario 1 desea realizar la instalación de un paquete A que depende de B y C; y un usuario 2 que quiere instalar un paquete D que depende de A. Para el usuario 1, su paquete principal será el A, mientras que sus dependencias serán B y C; mientras que para el usuario 2, el paquete será D y la dependencia A, que era el paquete que el usuario 1 quería instalar.

  Además, el usuario 2 no solamente tendrá que instalar D y A; sino que como A depende de B y C, también deberá instalarlas, por lo que en realidad para el usuario 2, el paquete será D y las dependencias A, B y C. Esto da origen a lo que se denomina árbol de dependencias, que sería el grafo que representa las relaciones de un paquete con sus dependencias. En la figura 3 se muestra el árbol dependencias del paquete D. Como se tratará más adelante, las dependencias supondrán uno de los mayores retos en el diseño de un sistema de gestión de paquetes, puesto que al construir el árbol de dependencias cuando se instalan varios paquetes, puede que haya en la lista de dependencias paquetes que no sean compatibles entre sí y el programa debe de decidir qué acción tomar en dichos casos.

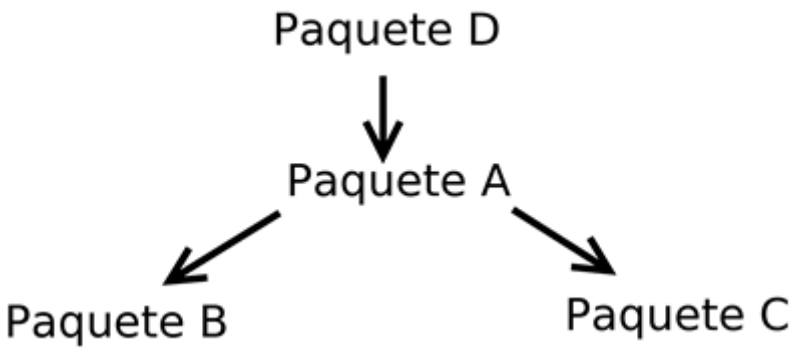

Figura 3: Ejemplo de árbol de dependencias.

Figura 4: Esquema de funcionamiento de un gestor de paquetes. Fuente: *Opensuse* [2].

- Repositorio: Es el lugar en el que se almacena y mantiene la información sobre los distintos paquetes. Esta ubicación puede ser tanto local, en un disco duro, o mediante un medio físico como *CD* o *DVD*, aunque es muy común que se encuentre en algún servidor, ya sea en una red de área local o en Internet, para que todos los usuarios de la red puedan acceder al contenido.
- Metadatos: Son ficheros, que también se encuentran en los repositorios, que proporcionan la información necesaria para instalar el paquete, así como información adicional sobre las características del mismo. Algunos de los campos relevantes que puede contener este fichero son la versión del paquete, la *URL* del repositorio en el que se encuentre, las dependencias del paquete, la suma de verificación de los contenidos, etc. En general, la cantidad de información de estos ficheros dependerá de la aplicación.

En la figura 4 se muestra el esquema del trabajo realizado por el gestor. Para conocer más detalle sobre el funcionamiento de los sistemas de gestión de paquetes, se analizarán los casos de los dos sistemas de paquetes más conocidos que utilizan distribuciones *GNU/Linux*: *APT* (*Advanced Package Tool*) [3], utilizando en distribuciones *Debian* y *YUM* utilizado en sistemas *Linux* basados en *RPM* (*Red Hat Package Manager*) [4].

2.1. *APT: Advanced Package Tool*

Inicialmente, únicamente existían los ficheros `.tar.gz`, que debían ser compilados cada vez que se querían usar en un sistema *GNU/Linux*. Cuando se creó *Debian*, se creó el primer sistema de gestión de paquetes llamado `dpkg`, que sentó las bases sobre el tema. Sin embargo, esta primera aproximación no cubría todas las necesidades y los creadores de *GNU/Linux* necesitaban conseguir un programa que realizaba la instalación de forma rápida, sencilla y eficiente, y que tratara automáticamente las dependencias. De aquí surgió el paquete que trataremos en esta sección, que es *APT*.

APT es uno de los sistemas de paquetes más conocidos creado por el proyecto Debian. Una de sus características principales, que lo hace avanzado es su aproximación a paquetes. *APT* no evalúa cada paquete de forma individual, sino que considera al conjunto para conseguir la mejor combinación de paquetes dependiendo de lo que esté disponible y además sea compatible, de acuerdo a las dependencias. [5]

2.1.1. Fichero de fuentes (*Sources list*)

Para que el programa pueda funcionar necesita un archivo que indique dónde se encuentran los paquetes, es decir en qué repositorios se encuentran. Este archivo es `/etc/apt/sources.list`. El formato que sigue este archivo es el siguiente [6]:

```
deb http://host/debian distribucion seccion1 seccion2 seccion3
deb-src http://host/debian  distribucion seccion1 seccion2 seccion3
```

El primer campo indica el tipo de fuente:

1. "`deb`" para los paquetes binarios: Estos paquetes pueden ser descomprimidos mediante los comandos `ar`, `tar` y `gzip` (en ocasiones también `xz` o `bzip2`) y contienen tres archivos:

   a `debian-binary`: Indica únicamente la versión del fichero `.deb` (actualmente se utiliza la versión 2.0).

   b `control.tar.gz`: Contiene los metadatos del paquete. Parte de la información sirve para determinar si el paquete se puede instalar o desinstalar de acuerdo a la lista de paquetes.

   c `data.tar.gz`: Contiene todos los datos (ejecutables, documentos...) del paquete. Este fichero puede tener otros formatos de compresión y podría ser `data.tar.bz2` para `bzip2`, `data.tar.xz` para *XZ* o `data.tar.lzma` para *LZMA*.

2. "`deb-src`" para los paquetes fuente. El contenido es similar al del `.deb`, salvo que ahora son los ficheros fuente originales, en lugar de los ficheros binarios. Un fichero `.deb-src` contiene tres ficheros: El `.dsc` (*Debian Source Control*), que es un fichero firmado, y describe el paquete, así como los ficheros que forman parte del mismo. Otro fichero es el `.origin.tar.gz`, que contiene el código original proporcionado por el desarrollador; y por último el `.debian.tar.gz`, que contiene las modificaciones realizadas sobre los

ficheros originales y los ficheros creados por el paquete *Debian* que contienen las instrucciones para construir el paquete.

El segundo campo indica la *URL* de la fuente. La *URL* puede comenzar por `file://` para indicar un fichero local, `http://` para indicar que los archivos están disponibles desde un servidor web, `ftp://` si los ficheros están en un servidor *FTP*, o `cdrom://` para instalaciones a partir de un disco de tipo *CD-ROM*, *DVD-ROM* o *Blu-ray*. Los últimos campos indican la estructura del repositorio. Si los paquetes están directamente en el lugar donde especifica la *URL*, simplemente se puede usar un "`./`", pero normalmente tendrán una estructura basada en distribuciones y secciones como usa *Debian*, y será necesario indicar la distribución, o suites (estable, en pruebas, inestable) y las secciones correspondientes. [5].

2.1.2. APT-GET

Dentro de *APT*, hay distintos programas para poder realizar las operaciones con los paquetes (*APT* no es un programa en sí mismo). El más utilizado es `apt-get`, que funciona mediante la línea de comandos, aunque también son de uso común otros con interfaz gráfica como `synaptic` o `aptitude`. El uso de `apt-get` suele ir también conjunto de `apt-cache`, que permite mostrar la información almacenada en la base de datos de *APT*. En esta sección analizaremos las principales opciones que ofrece `apt-get`.
En primer lugar, para poder realizar cualquier operación, es necesaria la lista de fuentes que se trató en la sección anterior. Sin embargo, es posible que esa lista contenga referencias que ya no existan. Para mantener la lista actualizada, existe la opción `update` (se usaría `apt-get update`).

Una vez que está disponible la lista de repositorios, el usuario se puede centrar en la verdadera funcionalidad del programa: la instalación, o en su defecto, desinstalación. Para ello existen los comandos `apt-get install package` y `apt-get remove package`. En ambos se comprueba automáticamente la lista de dependencias y se instalan/desinstalan los paquetes junto con los paquetes de los que dependen. También existe el comando `apt-get purge package` que realiza una desinstalación en la que también se eliminan los ficheros de configuración. Observe el siguiente ejemplo de instalación [6].

```
# apt-get install nautilus
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
bonobo libmedusa0 libnautilus0
The following NEW packages will be installed:
bonobo libmedusa0 libnautilus0 nautilus
0 packages upgraded, 4 newly installed, 0 to remove and 1 not upgraded.
Need to get 8329kB of archives. After unpacking 17.2MB will be used.
Do you want to continue? [Y/n]
```

En este ejemplo, al instalar el paquete *nautilus*, se necesita también instalar los paquetes *bonobo*, *libmedusa0* y *libnautilus0*, que son las dependencias de dicho paquete.

Para actualizar los paquetes a versiones más recientes, existe el comando `apt-get upgrade`. Cuando se ejecuta este comando, se buscan los paquetes instalados que pueden ser actualizados sin tener que borrar ningún paquete. En general, se buscará el paquete con el número de versión más alto (excepto para paquetes en estado experimental) Si se especifica en el `source.list`, *Testing* o *Unstable*, se puede conseguir que la actualización se haga a versiones de pruebas o inestables, lo cual debe hacerse con bastante precaución. Adicionalmente, la opción de actualización permite especificar si se quiere siempre actualizar a un tipo de distribución (ej.: siempre a una versión estable). Para ello se puede utilizar la opción `-t` (ej. `apt get -t stable upgrade`).

APT también permite gestionar las prioridades relacionadas con el origen de cada paquete. Para ello, se puede asignar una prioridad distinta a cada paquete disponible, de modo que *APT* siempre escogerá aquella versión con la prioridad más alta (salvo que la versión sea anterior a la instalada o tenga una prioridad inferior a 1000). Según el sistema de prioridades, cada versión instalada tiene una prioridad de 100, una versión no instalada, una de 500, que podría pasar a 990 en caso de estar en la distribución destino. Las prioridades de un archivo se pueden modificar en el archivo `/etc/apt/preferences`. Las premisas utilizadas son:

- Se instalará el paquete con mayor prioridad que cumpla las restricciones.
- En caso de empate en cuanto a prioridad, se escogerá el más reciente.
- Si dos paquetes tienen la misma versión y prioridad, pero distinto contenido, se elegirá aquel que no esté instalado.
- Un paquete con prioridad inferior a 0 nunca será instalado.
- Un paquete con prioridad entre 100 y 500, solo se instalará si no existe otra versión mayor instalada o que se pueda instalar utilizando otra distribución.
- Un paquete con prioridad comprendida en el rango de 501 a 990 se instalará si existe una versión mayor instalada o que se pueda instalar usando la distribución destino.
- Si la prioridad está comprendida entre 990 y 1000, el paquete se instalará siempre que la versión instalada sea menor.
- Si la prioridad es superior a 1000, se procederá con la instalación independientemente de la versión instalada.

2.1.3. Verificación de paquetes

Uno de los principales problemas de las redes actuales es la seguridad. Cuando un usuario utiliza *APT* para instalar un paquete, debe ser posible garantizar de alguna manera que ese paquete proviene de quién realmente lo distribuye y contiene la información que realmente el usuario quiere; y no un código malicioso que un atacante puede haber introducido para que se ejecute al instalar el paquete, con los perjuicios para el usuario, ya sea desvelando contraseñas o el daño con el que se haya diseñado el ataque. Para evitar este problema, se utiliza la firma digital.

El archivo que se encarga de proveer la información necesaria para la autenticidad es el *Release*, que es el que se firma. Este contiene un listado de los archivos *Packages*, que contienen la lista de los paquetes disponibles en la réplica que se utilice junto con sus *hashes*, lo que garantiza que el contenido de los paquetes no ha sufrido modificaciones. El programa `apt-key` se encarga de mantener el conjunto de claves públicas *GnuPG* (*GPG*) que se utilizan para verificar las firmas.

2.1.4. Actualización de paquetes

Otra de las principales funcionalidades de *APT*, es que permite actualizar los paquetes actualizados a versiones más recientes. Este proceso puede realizarse automáticamente en `apt-get` mediante el comando `apt-get upgrade`, que actualizaría todos los paquetes de la distribución actual. También se podría actualizar a otra nueva distribución mediante `apt-get dist-upgrade`.

Este proceso, aunque pueda parecer sencillo, puede acarrear ciertos problemas, que ocasionen que un paquete no se actualice aun disponiendo de una versión más reciente. Esto podría ocurrir si el paquete es una dependencia de otro paquete que no tiene una versión nueva para actualizar, en cuyo caso, para mantener las mismas dependencias del paquete principal se opta por mantener las versiones actuales. Otra situación es la aparición de nuevas dependencias, que podrían solicitar la versión anterior de un paquete instalado, por lo cual se prefiere no realizar cambios. Cuando un paquete no se actualice, se puede intentar pedir su instalación de forma individual mediante `install` para averiguar los problemas de dependencias que pueda ocasionar.

Uno de los posibles problemas de actualizar un paquete es que una vez actualizado no sea compatible con lo anterior, o que contenga errores dado que se encuentre en fase de pruebas al ser un *software* muy reciente. Para evitar estos problemas, existe un paquete llamado *apt-listchanges* que muestra información sobre los distintos problemas al inicio de la actualización para que el usuario pueda decidir la acción a tomar.

2.1.5. Interfaces: *Aptitude*, *Synaptic*

Ahora que ya conocemos las principales funcionalidades de *APT*, se mostrarán un par de interfaces de usuario, que facilitan el uso de la instalación.

Por una parte, tenemos *aptitude*, que se utiliza desde la consola, con un modo semigráfico. En el programa aparece una lista de los paquetes instalados y disponibles, y se muestra una descripción sobre el paquete, las versiones instaladas y disponibles, advertencias sobre la instalación, etc. Mediante la navegación a través de la interfaz se pueden señalar los paquetes que se desean instalar, eliminar o la operación que se desee realizar con ellos. En la figura 5 se muestra una captura de la interfaz que proporciona *aptitude*.

Aptitude muestra todos sus paquetes siguiendo una estructura en forma de árbol, en la que presenta distintos bloques de paquetes según su estado (actualizables, nuevos, instalados,

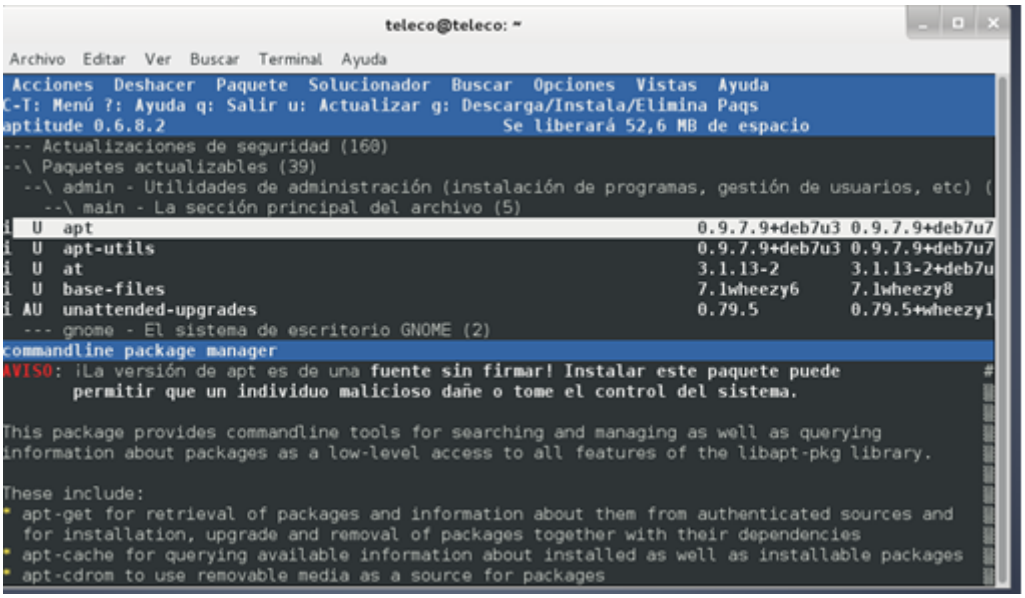

Figura 5: Gestor de paquetes *aptitude*.

no instalados y virtuales) y a través de cada categoría se puede ir desplegando e ir accediendo a cada uno de los paquetes individuales para acceder a su información y poder marcar para realizar una operación (se puede utilizar + para marcar para instalar, - para marcarlo para eliminar, _ para purgar, u para actualizar la lista de paquetes disponibles, `Shift+u`, para realizar una actualización global del sistema, etc...).

Por otro lado, otro gestor gráfico es *synaptic*, que provee una completa interfaz gráfica basada en *GTK+/GNOME*, y permite la instalación de forma más sencilla. En su interfaz, muestra un panel principal con los paquetes, de los que se muestra en un recuadro si está instalado o no, el nombre del paquete, su versión instalada, la más reciente y una descripción. También se permiten utilizar filtros para seleccionar grupos de paquetes, por estado, temática origen, o arquitectura y en conjunto con filtros de búsqueda que puede establecer el propio usuario.

Synaptic también permite el marcado de paquetes para las distintas operaciones. Haciendo clic en un paquete con el botón derecho o mediante el panel "Paquete" se puede marcar un paquete para instalar, desinstalar, reinstalar, actualizar o desinstalar completamente (purgar) y una vez que se hayan realizado todas las marcas, se puede utilizar la opción "Aplicar" para realizar todos los cambios efectuados. En la figura 6 se muestra una imagen de la interfaz de *synaptic*.

2.2. *YUM: Yellow Dog Updater Modified*

Si bien, se ha analizado el comportamiento de *APT* para las distribuciones *Debian*, ahora se realizará lo propio con *YUM*, el sistema de gestión de paquetes para sistema *Linux* basados en *RPM*. La filosofía del mismo es similar a la de *APT*: *YUM* se encarga de instalar, desinstalar y actualizar paquetes tratando los posibles conflictos entre dependencias. *YUM* ini-

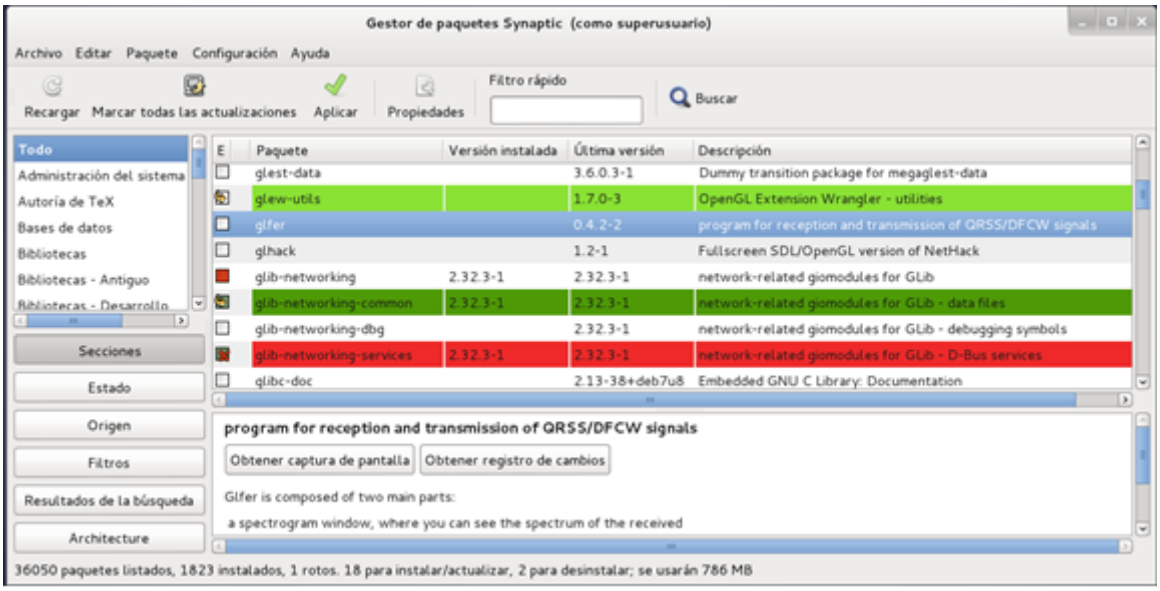

Figura 6: Gestor de paquetes *synaptic*.

cialmente obtiene las cabeceras del paquete para analizarlas (los ficheros de metadatos que contienen son `primary.xml.gz`, que contiene la información principal; `filelists.xml.gz`, que describe los ficheros que se van a instalar y algunos datos generales sobre el nombre, versión y arquitectura del paquete; `repomd.xml`, que contiene información sobre los archivos del repositorio, y `other.xml.gz`, que completa la información con el autor, detalles de la versión, errores corregidos...) y tomar decisiones; y cada vez que descarga cabeceras o el propio paquete, lo realiza también sobre repositorios a los que se puede acceder mediante *FTP*, *HTTP*, un servidor *NFS* o un sistema local de archivos. El programa mantiene un directorio `/var/cache/yum` para almacenar las cabeceras y archivos descargados [7] (en *APT* este directorio es `/var/cache/apt/archives`).

El fichero de configuración se encuentra en `/etc/yum.conf`. Este fichero siempre contiene una sección `[main]`, que permite establecer opciones de carácter global y también contiene una o más secciones `[repository]` para especificar opciones específicas de repositorios; aunque se recomienda definir estar opciones en ficheros `.repo` en el directorio `/etc/yum.repos.d`. Algunas de las opciones que se definen aquí afectan al modo de operar de `yum` o al modo de tratar los repositorios (se puede establecer el nivel de mensajes que muestra el programa, activar comprobaciones de firmas de los paquetes, cambiar el directorio caché por defecto,...). Un ejemplo de fichero `yum.conf` es el siguiente: [8]

```
[main]
cachedir=/var/cache/yum/$basearch/$releasever
keepcache=0
debuglevel=2
logfile=/var/log/yum.log
exactarch=1
obsoletes=1
gpgcheck=1
plugins=1
installonly_limit=3
```

```
[comments abridged]

# PUT YOUR REPOS HERE OR IN separate files named file.repo
# in /etc/yum.repos.d
```

2.2.1. Opciones principales de *YUM*

YUM, al igual que `apt-get` dispone de muchas opciones. Aquí se presentarán algunas de las más importantes.

Por un lado, es posible mostrar una lista de todos los paquetes disponibles en cualquier repositorio. Para ello existe la opción `list`. Si no se utiliza ningún argumento (es decir, el comando sería `yum list`), se mostrarían todos los paquetes disponibles e instalados; aunque también se pueden incluir opciones para mostrar únicamente los paquetes disponibles para instalar, que no están instalados (`available`), solo los paquetes instalados (`installed`), paquetes que tienen actualizaciones (`updates`), etc. También es posible buscar por algún paquete en concreto escribiendo el nombre del paquete (ej. `yum list tsclient`).

Si no se conoce siquiera el nombre del paquete, *YUM* también proporciona una opción para buscar (`search`) en el que se puede introducir una cadena de texto, que será buscada en los nombres, descripciones, resúmenes y listas de paquetes para encontrar paquetes que tengan algún campo en el que coincida dicha cadena.

Entre las opciones relacionadas con las operaciones principales, al igual que en `apt-get` existe la opción `install`, para instalar nuevo software, `update`, para instalar nuevo software (en lugar de `upgrade`), `remove` para desinstalar un paquete, `localinstall`, para instalar un paquete, que se encuentra en el sistema local de archivos, en lugar de descargarlo de un repositorio; `reinstall`, para reinstalar un paquete, etc. Un ejemplo de instalación es el siguiente [9]:

```
# yum install postgresql.x86_64
Resolving Dependencies
Install       2 Package(s)
Is this ok [y/N]: y

Package(s) data still to download: 3.0 M
(1/2): postgresql-9.0.4-5.fc15.x86_64.rpm          | 2.8 MB     00:11
(2/2): postgresql-libs-9.0.4-5.fc15.x86_64.rpm     | 203 kB     00:00
----------------------------------------------------------------------
Total                                        241 kB/s | 3.0 MB     00:12

Running Transaction
  Installing : postgresql-libs-9.0.4-5.fc15.x86_64             1/2
  Installing : postgresql-9.0.4-5.fc15.x86_64                  2/2

Complete!
```

2.2.2. Conclusiones

A la vista de las opciones principales de *APT* y de *YUM*, se aprecia una gran similitud entre los dos sistemas de gestión de paquetes, ya que prácticamente las opciones que ofrecen son las mismas. Algunas diferencias, como que *YUM* actualiza la lista de paquetes automáticamente y que *APT* necesita la opción `update`, o que la actualización para *apt-get* sea `upgrade` y para *YUM* `update`, son mínimas en cuanto al comportamiento general del programa. Respecto a la versión gráfica, al igual que existe *synaptic* para APT, también *YUM* dispone de *YumEx* (*Yum Extender*).

Las diferencias más notables se habrán podido percibir cuando se describieron los archivos de configuración, y es que más puede diferenciar a los dos gestores, son los formatos de los metadatos que utilizan para obtener la información de la instalación. Es por ello, que este será uno de los aspectos que más se tendrán que cuidar en el proceso de creación del sistema de gestión de paquetes.

2.3. *OneGet*

Antes de concluir este capítulo, se presentará el nuevo sistema de gestión de paquetes, *OneGet*. Durante toda la discusión, siempre se ha tratado de sistemas de distintas plataformas *Linux*, pero no se había tratado nada de *Windows*, que es el sistema operativo con más cuota de mercado. La cuestión es que hasta ahora *Windows* no había tenido su propio gestor de paquetes y los únicos instaladores provenían de terceros, pero en el desarrollo del próximo sistema operativo, *Windows 10*, cuyo lanzamiento se prevé que se realice este año, se ha incluido por primera vez un sistema de gestión de paquetes: *OneGet*.

El nuevo *OneGet* será parte de *PowerShell*. *Windows PowerShell* es un lenguaje de *scripting* y *shell* de línea de comandos diseñado con especial interés para la administración [10]. Permite crear *scripts* para realizar funciones llamados *cmdlets,* que son programados utilizando `C#` y que se ejecutan en el entorno de *PowerShell*.

Las distintas acciones que *OneGet* realizará estarán llevadas a cabo por distintos *cmdlets* (existirá un *cmdlet* para instalar, otro para desinstalar, y así para cada acción principal). La figura 7 muestra cómo serán estas opciones.

Una de las ventajas que tendrá es la posibilidad de crear repositorios con el *software* que un desarrollador quiera, de modo que mientras que *Microsoft* mantendrá su propio repositorio con sus propios programas, cualquier desarrollador podrá crear su repositorio para que sus usuarios puedan instalar sus programas fácilmente (este hecho hace que no exista un fichero de fuentes fijo, que cada fichero de metadatos tenga su propio nombre y que el directorio caché exacto pueda variar).

La instalación será también sencilla. Simplemente habrá que indicar el nombre del paquete y automáticamente el programa se descargará e instalará, sin necesidad de tener que descargar ningún instalador y ejecutar el ejecutable (`.exe`), lo cual puede resultar más cómodo para el usuario, sobre todo porque se podrán evitar las diversas pantallas en las que al pulsar

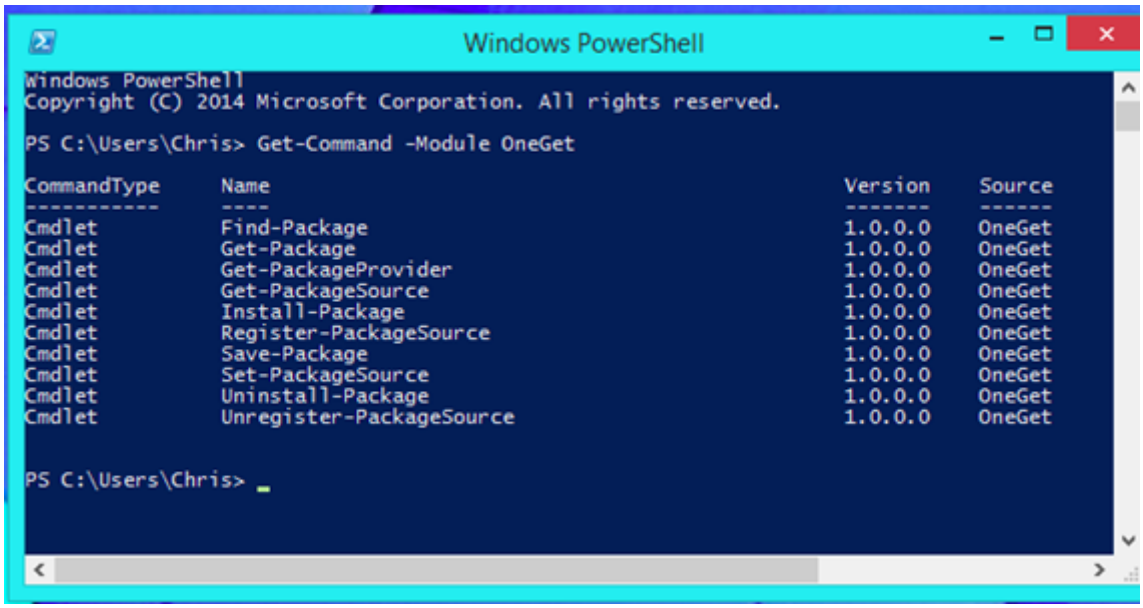

Figura 7: Distintas opciones de *OneGet*. Fuente: *How-to Geek* [11].

Aceptar, se pueda instalar algún software no deseado durante una instalación. De momento existe también la opción de desinstalar, aunque no hay todavía soporte para actualizar el software a la última versión, tal y como vimos en *APT* o en *YUM*. Para instalar un programa el comando será `Install-Package` (ej. `Install-Package -Name Firefox`).

Entre la posible aplicación de este programa, es que incluso podría integrarse en *Windows Store* para la distribución de aplicaciones. De momento, el sistema está en pleno desarrollo, pero es posible que se convierta en un elemento fundamental del sistema operativo.

Tras tratar este gestor de paquetes, es de notar, que al igual que los dos anteriores, las funcionalidades e incluso la sintaxis de los comandos es muy parecida, pero lo que cambia por completo es la arquitectura, al aparecer los `cmdlets`, que no aparecían en *Linux*. Por ello, la elección de la misma, será uno de los puntos clave para el desarrollo del gestor de paquetes. Para concluir este aparatado, la tabla 1 muestra un resumen de las características de cada uno de los gestores de paquetes analizados.

Característica	*APT*	*YUM*	*OneGet*
Sistema operativo	*Linux*	*Linux*	*Windows*
Versión por línea de comandos	*apt-get*	*yum*	*OneGet*
Versión gráfica	*Synaptic*	*YumEx*	-
Fuentes	`/etc/apt/sources.list`	`/etc/yum.conf`, `/etc/yum.repos.d`	Depende del repositorio
Metadatos del paquete	`control.tar.gz`	`primary.xml.gz`, `filelists.xml.gz`, `other.xml.gz`, `repomd.xml`	Depende del paquete (en general se usan ficheros `.ps1`)
Directorio caché	`/var/cache/apt/archives`	`/var/cache/yum`	En distintos directorios dentro de `AppData\Local` dependiendo del paquete
Actualizar la lista de paquetes	`Update`	-	-
Instalar	`Install`	`Install`	`Install-Package`
Desinstalar	`Remove`	`Remove`	`Uninstall-Package`
Actualizar	`Upgrade`	`Update`	-

Tabla 1: Resumen comparativo de los distintos gestores de paquetes.

3. Definición de los metadatos

El primer paso del desarrollo del proyecto, antes de comenzar a programar ninguna de las opciones que permitirá el programa, será definir la estructura de metadatos.

El formato de datos que se elegirá para todos los ficheros será el `.json`, dado que es un formato muy adecuado para el intercambio de datos y además su estructura tiene forma de diccionario, lo cual puede ser muy útil para la programación posterior, puesto que se podrá establecer una correspondencia directa entre el contenido del fichero y el de un diccionario. Algunas alternativas al `.json` bastante comunes son el `.xml`, pero dado que no aporta una clara ventaja respecto al anterior y su estructura de datos, aunque se podría *parsear* y convertir en un diccionario, no sería tan evidente como en el caso de un fichero `.json`.

Con respecto al lenguaje de programación, si bien todavía no es relevante para definir el formato de los metadatos, sí que se puede plantear su discusión.

La versión inicial del programa, llamado *myum* (de Microelectronic *YUM*), estaba desarrollado en `TCL` (*Tool Command Language*), que es un lenguaje interpretado. El extendido uso de este lenguaje dentro de la sección se debe en parte, a las ventajas que proporciona en el control de simulaciones de código *VHDL* de descripción de hardware, que es una de las actividades principales.

Para la nueva versión, sin embargo se planteó la opción de cambiar de lenguaje. Dentro de los lenguajes usados en la sección, *Python* se mostró como la alternativa a *TCL*. Entre las ventajas que disponía es que dispone de un soporte mucho mayor y dispone de más librerías que podrían facilitar mucho el trabajo y además, podrían permitir una mayor extensibilidad en el futuro, pues el uso de un módulo de *Python* puede dar más funcionalidad de la que inicialmente se puede requerir. También este lenguaje permite realizar de forma más eficiente proyectos complejos.

Entre las desventajas, se podría citar el hecho de que no sería posible reutilizar ninguna parte del código ya existente. Sin embargo, como el *myum* inicial no es funcional, tampoco supone una gran ventaja, y dado que hay que reescribir el programa de nuevo, dada las ventajas presentadas, es una mejor solución utilizar *Python* desde el inicio y aprovechar todo lo que el lenguaje ofrece para la solución del problema.

Respecto a las versiones de *Python*, en *Windows* se dispone de *Python 3.4* y *Python 2.7*, y en las máquinas *Linux* y *Cygwin* [12], en general se tiene *Python 2.7* y en algunos casos también *Python 3.2*, por lo que para el correcto funcionamiento independiente del lugar donde se ejecute el programa, la programación deberá tener en cuenta, no solo los problemas de incompatibilidad entre plataformas, sino entre las distintas versiones del intérprete de *Python*.

3.1. Formato de fuentes

La primera de las decisiones será establecer el modo en el que el programa conocerá dónde se encuentra cada paquete.

```
{
    "json_database_version": "2.0.0",
    "repositories": [
        {
            "name": "adc",
            "path": "git@vmgitlab:microelectronic_crip/crip_adc.git"
        },
        {
            "name": "agent_adc",
            "path": "git@vmgitlab:microelectronic_crip/crip_adc.git"
        },
        {
            "name": "agent_apb",
            "path": "git@vmgitlab:microelectronic_crip/crip_apb.git"
        },
        {
            "name": "agent_clkrst",
            "path": "git@vmgitlab:microelectronic_crip/crip_setup.git"
        },
        {
            "name": "agent_common",
            "path": "git@vmgitlab:microelectronic_crip/crip_setup.git"
        }
    ]
}
```

Figura 8: Alternativa 1 de formato de fuentes.

Dentro de *VMGitLAB*, existe un repositorio para cada *IP* (llamado *crip*, anteponiéndole las dos primeras iniciales de la empresa) y dentro de este se sitúan diferentes paquetes.

Un posible formato sería mantener la lista de todos los paquetes individuales y la ruta del repositorio donde se encuentran y a la que habría que acceder para obtener el paquete. La figura 8 muestra dicho formato.

Entre las ventajas e inconvenientes de este formato podemos destacar las siguientes:

VENTAJAS:

- Se conoce directamente la *URL* del paquete, por lo que si se pide su instalación, no habrá nada más que acceder al *path* indicado en este fichero.

INCONVENIENTES:

- Si se añade un nuevo un paquete a un *crip*, éste no estará disponible a menos que se modifique este fichero.
- No tiene soporte para habilitación e inhabilitación de repositorios.

Otra de las alternativas es mantener el formato que disponía la versión inicial del *myum*, en el

```
{
    "json_database_version" : "1.0",
    "repositories" : [
        {
         "name"     : "crip_technology",
         "access"   : "ssh",
         "server"   : "vmgitlab:",
         "username" : "git",
         "path"     : "microelectronic_crip/crip_technology.git",
         "enable"   : "no"
        },
        {
         "name"     : "crip_sdebug",
         "access"   : "ssh",
         "server"   : "vmgitlab:",
         "username" : "git",
         "path"     : "microelectronic_crip/crip_sdebug.git",
         "enable"   : "no"
        }
    ]
}
```

Figura 9: Alternativa 2 de formato de fuentes.

cual aparecían directamente los repositorios de los *crips*, en lugar de los paquetes individuales. Este formato se muestra en la figura 9

Los aspectos positivos y negativos de este formato son:

VENTAJAS:

- Se utiliza directamente el *crip*, en lugar del paquete, lo que mejora la escalabilidad, ya que si se introduce un nuevo paquete dentro del *crip*, se podría instalar sin necesidad de modificar el fichero.
- Permite habilitar y deshabilitar repositorios mediante la clave `'enable'`.

INCONVENIENTES:

- Sistema para encontrar la *URL* demasiado complejo. En primer lugar, diferencia en tipo de acceso; solo admitía `"ssh"` para repositorios remotos, en cuyo caso, la *URL* se construía como `username@serverpath` (en este caso `git@vmgitlab:microelectronic_crip/crip_sdebug.git"`) y para repositorios locales usaba la sintaxis `file://path`.
- Necesita procesado adicional para obtener la *URL* de un paquete conociendo únicamente las direcciones de los repositorios.

El primero de los inconvenientes es sencillo de subsanar, puesto que la primera alternativa incluía únicamente un campo `'path'`, en el que se indicaba la dirección fuera del tipo que fuere, lo que evitaba cualquier tipo de procesamiento adicional.

```
{
    "json_database_version": "2.0.0",
    "repositories": [
        {
            "name": "crip_adc",
            "path": "git@vmgitlab:microelectronic_crip/crip_adc.git ",
            "enable": "yes"
        },
        {
            "name": "crip_apb",
            "path": "git@vmgitlab:microelectronic_crip/crip_apb.git ",
            "enable": "yes"
        },
        {
            "name": "crip_setup",
            "path": "git@vmgitlab:microelectronic_crip/crip_setup.git ",
            "enable": "no"
        }
    ]
}
```

Figura 10: Alternativa 3 de formato de fuentes.

Sin embargo, el último inconveniente deja ver que hay dos formas opuestas de plantear este problema, que ambas tienen ventajas e inconvenientes. Las dos alternativas son que aparezcan en el fichero todos los paquetes y sus direcciones, o los repositorios de los *crips* y sus direcciones. En cuanto a procesamiento la mejor opción es la primera, porque hay una relación directa entre paquete y *URL*. Sin embargo, la segunda tiene mejor escalabilidad. Como el cómputo que hay que realizar es mínimo (solo hay que comprobar para cada paquete el repositorio al que pertenece y de ahí obtener su *URL*), se prefiere mantener la lista de repositorios y dar prioridad a la escalabilidad.

De este modo, se presenta una tercera alternativa basada en la segunda (muestra las direcciones de los repositorios), que mejora los problemas de las direcciones con lo expuesto en la primera. El formato de esta alternativa se muestra en la figura 10.

En principio, este fichero será estático en una ubicación determinada, si bien se actualizará manualmente con la aparición de nuevos repositorios, o bien, se podrá hacer una copia del mismo y modificarla libremente para que después el *myum* utilice este nuevo fichero.

Una de las alternativas que se plantearon fue que se ejecutase un código en el servidor que cada cierto tiempo (unos 5 minutos) comprobase los repositorios actuales, de modo que si se añade un nuevo repositorio, se añadiese automáticamente, no como ahora, que si se añade un paquete dentro de un repositorio que ya aparece en la lista, sí lo podrá usar el programa, pero si aparece un *IP* totalmente nuevo, al no estar en la lista no podrá ser usado.

Esta idea, a pesar de la ventaja aparente, tiene el gran inconveniente de que aumenta la carga del servidor, y si en las condiciones actuales, ya se observan problemas de funcionamiento cuando hay sobrecarga, esto podría llevar a un peor funcionamiento. Por otra parte, si el servidor crea una lista con los repositorios, solo podrán aparecer los repositorios que se

encuentren en el servidor *VMGitLab*, por lo que si en el futuro se realizase una migración o se quisiesen utilizar *IPs* de otros centros del grupo *Airbus*, al que pertenece la empresa, este sistema no daría la escalabilidad necesaria; mientras que si se mantiene el fichero estático, si bien, requeriría modificaciones ante nuevos repositorios, permite la aparición de cualquier dirección; y la aparición de nuevos *crips* no es algo demasiado frecuente, por lo que incluir una entrada al fichero de forma muy esporádica tampoco supone un gran problema. Si se utilizase el formato de incluir paquetes en lugar de *crips*, sí que daría mayor ventaja porque paquetes nuevos pueden aparecer con mayor frecuencia, pero para este caso no, por lo que finalmente se desestima esta alternativa.

3.2. Nomenclatura de versiones

Antes de definir la estructura de metadatos de los distintos paquetes, es necesario plantearse cómo se definirán las versiones. Hasta ahora, la sección utilizaba una nomenclatura de tipo a.b (es decir, existen versiones 1.0, 1.1, 1.2).

En principio, este formato no parece problemático, salvo cuando en los repositorios aparecen de un mismo paquete versiones 0.9, 0.91, 0.92 y así sucesivamente. Esto es un claro problema, puesto que si existe una versión 0.12 y otra 0.92, ¿cuál de ellas es más reciente? En teoría, 92 es mayor que 12, y por tanto la 0.92 sería más reciente, pero si cuando se implementó la versión 0.9, se hicieron cambios y surgió la versión 0.91 y 0.92 a posteriori, y la 0.12 es la 12ª dentro de las que tienen como primer dígito el 0, la 0.12 sería más reciente. Pero es que tampoco hay seguridad de que la 0.12 no sea un pequeño cambio respecto de la 0.1.

Si el gestor de paquetes tiene una función para actualizar paquetes, de este modo no habría forma de aclararse. Una opción será hacer que el primer dígito después del punto representase cambios menores de la versión principal, y el segundo dígito fueran correcciones de la misma, de modo que quedaría claro que la versión 0.92 sea siempre más reciente a la 0.12. Sin embargo, de nuevo existe un problema, y es que el número de versiones quedaría limitado, pues ya no podría existir una décima subversión, puesto que sería la 0.10, que tendría el mismo formato que la versión 0.1 inicial.

Para resolver este problema, una segunda alternativa es utilizar el versionado semántico [13]. Éste define un formato a.b.c, donde el significado de cada elemento es el siguiente:

a *Major*: Cambio significativo, que provoca que el código de una versión anterior no sea compatible.

b *Minor*: Cambio que modifica o añade alguna característica al código ya hecho, pero que no rompe compatibilidad con lo anterior.

c *Patch*: Cambio retrocompatible para arreglar algún error en el programa (*bug fix*).

Con este nuevo formato, las versiones anteriores quedarían cambiadas a 0.12.0, 0.9.0, 0.9.1 y 0.9.2, lo que evitaría cualquier tipo de ambigüedad en qué versión es más reciente y se

establecería una política más concreta de cuándo se establece cada tipo de versión según los cambios realizados, aspecto que anteriormente podría ser algo confuso.

3.2.1. Formato de tags

Al hilo con la nomenclatura de versiones, está el formato de los *tags*. Una vez que se tiene lista una versión, hay que subirla al repositorio para que esté pueda usarse para su instalación. Para separar cada una de las versiones, se establece un *tag*, que es un punto en el histórico de *commits* de *Git* que marca, en nuestro caso, el punto en el que una versión está terminada. De este modo según se van realizando modificaciones, el código en el repositorio cambiará pero el sistema de control de versiones será capaz de acceder al contenido que había previamente con su registro de cambios.

Previamente, el formato de los *tags* era `"tag_nombre_vversion"` (ej: `tag_pkg_v1.0`). En principio no existe ningún motivo por el cual este formato no sea válido, puesto que proporciona la información básica para reconocer a un paquete en una versión dada. El cambio principal es que ahora las versiones utilizarán versionado semántico, pero por el resto lo único que se puede plantear es qué hacer con los *tags* antiguos.

Una alternativa es adaptar el código del programa para adaptarse al formato antiguo y otra es empezar de nuevo el sistema de *tags* desde la versión 1.0.0 y obviar los *tags* pasados.

La primera de las alternativas, tiene la ventaja de que en cuanto al contenido de *Git*, en teoría no serían necesarios muchos cambios puesto que el *myum* sería capaz de interpretarlos. Sin embargo, lo que parecen pocos cambios quizás no lo sean tanto, porque si el formato de los paquetes o los ficheros de instalación cambian ya no serían válidos y habría que adaptarlos; y es más, toda la discusión sobre el formato de versiones se vendría abajo, puesto que aunque el *myum* trabajase con versionado semántico, sería necesario establecer reglas concretas para el comportamiento de aquellos paquetes que se mantienen en versiones antiguas.

El otro punto de comenzar de nuevo el sistema de *tags* muestra el principal inconveniente de tener que revisar el contenido de las versiones y adaptarlas al nuevo formato con nuevos *tags* para que funcionen, aunque si va a haber más cambios, al final esto va a ser necesario, por lo que tampoco se debe intentar evitar lo inevitable. Para diferenciar los *tags* antiguos de los nuevos, se debe establecer algún tipo de marca y se ha decidido anteponer la letra 'm' de microelectrónica a la palabra *tag*. También, una vez analizado el formato se ha llegado a la conclusión de que la `'v'` en versión no aporta nada, por lo que se ha optado por suprimirla.

De este modo, el formato de *tags* es `mtag_nombre_version` (ej. `mtag_pkg_1.0.0`).

3.2.2. Definición de los metadatos del paquete

El siguiente paso en la definición de los formatos con los que trabajará el gestor de paquetes será el de los metadatos del paquete. En principio, el contenido más importante es la localización de los ficheros de instalación dentro del repositorio y el lugar dónde deben ser

copiados tras la instalación; las dependencias del paquete y los comandos que se deben ejecutar durante la instalación.

De este modo, originalmente existe el fichero `myum_repo_content.json`, que incluye el contenido de todos los paquetes del mismo repositorio. La figura 11 muestra el formato de dicho fichero.

Este formato incluye todos los paquetes en todas las versiones que existen en el repositorio, de modo que por cada versión exista una entrada en el fichero. Esto genera el inconveniente, de que si crece el mucho el número de versiones, crecerá mucho el contenido del fichero aun habiendo mucha información repetida, puesto que en muchas ocasiones, los directorios que copiar o los comandos a ejecutar son los mismos en distintas versiones, aunque el contenido de los directorios sea distinto.

Por otra parte, no es de extrañar que cuando se termine de desarrollar un paquete, se realicen los *commits* y se olvide actualizar el fichero; y otro inconveniente es que si *myum* trabaja con paquetes, ¿por qué realizar una orientación a repositorios en los metadatos y no directamente a paquetes que es con lo que el programa trabaja?

Una mejora de este fichero es la separación de cada uno de los paquetes en ficheros de metadatos distintos y que cuando se acceda a ellos, se haga a través del *tag* de la versión, por lo que si en una nueva versión cambia el contenido, pero no el directorio donde se encuentra, no sea necesario realizar cambios en los metadatos, lo que facilita la tarea de mantener actualizados los repositorios. De este modo, el anterior `myum_repo_content.json` de ejemplo que contenía dos paquetes se desdoblaría en dos ficheros `pkg1.json` y `pkg2.json`. El contenido de uno de ellos se muestra en la figura 12.

De esta manera, si tenemos un repositorio con 4 paquetes y 4 versiones con paquete, pasaremos de tener un fichero con 16 entradas, a 4 ficheros con 1 entrada cada uno.

Sobre este formato, todavía hay aspectos que se pueden mejorar, y es que por ejemplo si ahora accedemos al fichero `pkg.json`, una vez que entremos al *tag*, que lleva la información de nombre y versión, el campo nombre y versión ya no son relevantes y se podrían suprimir.

Una de las propuestas en torno a la especificación de las dependencias fue cambiar la estructura de una lista de diccionarios a un único diccionario, teniendo en cuenta que el número de claves para cada dependencia es de 2, que se pueden resumir en un par clave/valor. De este modo, la estructura de dependencias del paquete `pkg1.json` se podría reducir a:

```
{
        "dep1": "v1.0",
        "dep2": "v2.0"
}
```

Donde evidentemente las versiones deberán ir en versionado semántico, pero de momento ese no es el problema. Esta alternativa tiene la ventaja de que permite una estructura más compacta, al evitar tener diccionarios dentro de una lista. Sin embargo, tiene un par de problemas fundamentales. El primero de ellos es que pierde la filosofía de un diccionario donde existen pares clave/valor, puesto que en teoría la clave debe ser conocida y a través de ella debe

```
{
    "json_database_version" : "1.0",
    "packages" : [
        {
            "name"    : "pkg1",
            "version" : "v1.0",
            "dependency" : [
                { "name" : "dep1", "version" : "v1.0" },
                { "name" : "dep2", "version" : "v2.0" }
            ],
            "data" : [
                { "type" : "dir",  "source": "source/design1", "dest": "dest/
                   design1" },
                { "type" : "file", "source": "source/design2", "dest": "dest/
                   design2" }
            ],
            "exec" : [
                {"cmd" : "cmd1"},
                {"cmd" : "cmd2"}
            ]
        },
        {
            "name"    : "pkg2",
            "version" : "v2.0",
            "dependency" : [
                { "name" : "dep3", "version" : "v3.0" },
                { "name" : "dep4", "version" : "v4.0" }
            ],
            "data" : [
                { "type" : "dir",  "source" : "source/design1", "dest": "dest/
                   design1" },
                { "type" : "file", "source" : "source/design2", "dest": "dest/
                   design2" }
            ],
            "exec" : [
                {"cmd": "cmd1"},
                {"cmd": "cmd2"}
            ]
        }
    ]
}
```

Figura 11: Contenido del fichero `myum_repo_content.json`.

```
{
    "json_database_version" : "1.0",
    "packages" : [
        {
            "name"    : "pkg1",
            "version" : "v1.0",
            "dependency" : [
                { "name" : "dep1", "version" : "v1.0" },
                { "name" : "dep2", "version" : "v2.0" }
            ],
            "data" : [
                { "type" : "dir",  "source": "source/design1", "dest": "dest/
                   design1" },
                { "type" : "file", "source": "source/design2", "dest": "dest/
                   design2" }
            ],
            "exec" : [
                {"cmd" : "cmd1"},
                {"cmd" : "cmd2"}
            ]
        },
    ]
}
```

Figura 12: Contenido del fichero `pkg1.json`.

conocerse su valor. Aquí a priori, no se puede conocer ni la clave ni el valor, mientras que antes las claves eran fijas "`nombre`" y "`versión`". Este problema no es el fundamental, porque mediante código se puede tratar la información que se desea igualmente, pero el segundo de los inconvenientes puede tener un mayor peso. Actualmente, de cada dependencia solo se indica su nombre y su versión y con un par clave/valor es suficiente para indicar ambos campos, pero si en el futuro apareciese otro nuevo campo relevante para identificar un paquete (ej. centro del que proviene), con esta alternativa no habría soporte para introducir nuevos campos y habría que reescribir gran parte del código para cambiar al nuevo formato. Sin embargo, manteniendo la estructura de la lista con diccionarios se permite introducir nuevos pares clave/valor para cada diccionario de forma sencilla, de forma que con pocos cambios en el código, se pudiera extender la funcionalidad. Por ello, esta alternativa, se descarta.

Típicamente, uno de los comandos que suele indicar en los paquetes es el "`chmod`", que sirve para cambiar los permisos. Este comando, sin embargo, no funciona en *Windows* y si se quiere aplicar a varios directorios separados, se debe ejecutar varias veces. *Python*, proporciona en sus librerías la funcionalidad para cambiar permisos y dentro de que el sistema de permisos de *Windows* es diferente al de *Linux* (*Windows* no tiene el sistema de permisos para propietario, grupo y resto de *Linux*), intenta que los permisos establecidos sean lo más parecido posible entre sus equivalentes. Por ello, una mejora es indicar en la sección de datos a transferir un campo llamado "`chmod`", en el que se indique el número octal de los permisos que se quieran asignar al fichero o directorio que se indique, siguiendo el criterio que usa *Linux* y que la rutina de *Python* se encargue de ajustar los permisos de acuerdo a ese valor de la mejor manera posible.

```
{
    "json_database_version": "2.0.0",
    "packages": {
        "data": [
            {
                "dest": "design/devices/source/crip_apb",
                "source": "design/devices/source/crip_apb",
                "type": "dir",
                "chmod": 755
            }
        ],
        "dependency": [{
            "name": "basic",
            "version": "1.0.0"
        }],
        "exec": []
    }
}
```

Figura 13: Formato del paquete *APB*.

Dentro de los tres dígitos en octal para cada versión el primero indica los permisos para el propietario, el segundo para los miembros del grupo y el tercero para el resto de usuarios. De entre los 8 valores que puede tomar cada dígito (del 0 al 7), el bit de lectura suma 4 al total, el de escritura 2 y el de ejecución 1 (ej. si se desean solo permisos de lectura el valor sería 4, si se desean permisos de lectura y ejecución, un 5, y así para cada caso). Con estos cambios, el formato definitivo para los paquetes es el que se muestra en la figura 13.

3.3. Definición del fichero de instalación

Cuando se va a desarrollar un proyecto, el desarrollador se clona el contenido del repositorio del proyecto de forma local y a partir de ahí empieza a trabajar. Cada proyecto requiere de ciertos paquetes para su funcionamiento, y estos son conocidos de antemano, por lo que una buena práctica es mantener para cada proyecto un fichero de instalación en el que se definan los paquetes necesarios, de modo que quien se ponga a desarrollar no tenga que preocuparse por qué paquetes necesita, sino que ya tenga una lista predefinida y simplemente ejecute el *myum* e instale todo lo necesario.

Este aspecto, es quizás una de las diferencias más significativas con respecto a un gestor de paquetes convencional, en el cual no hay distinción entre el paquete que se desarrolla y cualquier otro.

También es necesario marcar la diferencia entre los paquetes necesarios para hacer funcionar un paquete y los requeridos para poder desarrollarlo. Es posible que un paquete A dependa de B y de C, pero necesita para poder desarrollarse los paquetes D y E. De este modo un usuario que instale A, solo instalará con él B y C; pero aquel que desarrolle A, instalará B, C, D y E, pero no instalará A, porque se habrá clonado directamente el contenido del repositorio para trabajarlo.

```
{
    "json_database_version" : "1.0",
    "packages" : [
        {
            "name"    : "pkg1",
            "version" : "v1.0"
        },
        {
            "name"    : "pkg2",
            "version" : "v1.0"
        },
        {
            "name"    : "pkg8",
            "version" : "v1.0"
        }
    ]
}
```

Figura 14: Formato inicial del fichero `myum_install_def.json`.

Por ello, en el fichero `pkg.json` se indicaban las dependencias de un paquete y en el fichero de instalación (al que llamaremos `myum_install_def.json`) se indicarán los paquetes necesarios para desarrollarlo.

El fichero `myum_install_def`, aunque funcione dentro del ámbito del desarrollo de un paquete no tiene por qué limitarse a ello, y un usuario puede que necesite instalar ciertos paquetes podría definirse un fichero con dicho formato y realizar una instalación con el mismo.

Ahora que ya se conoce la necesidad del fichero de instalación, se intentará ver cuál es su formato más adecuado. El formato inicial de este fichero se muestra en la figura 14.

Las ventajas e inconvenientes que presenta son:

VENTAJAS

- Establece de forma clara y sencilla, los paquetes necesarios a instalar, indicando el nombre y la versión.

INCONVENIENTES

- Presenta toda la lista de paquetes, sin hacer distinción entre lo que son dependencias del paquete y lo que es necesario para el desarrollo del mismo.
- Obliga a indicar las dependencias, dato que se podría extraer directamente del fichero `pkg.json` que se encuentra en el mismo directorio que el `myum_install_def.json`.

Una manera de separar lo que son dependencias de los paquetes para el desarrollo es eliminar las entradas de las dependencias de este fichero e indicar como instalar el propio

paquete que se va a desarrollar, para que al crear el árbol de dependencias, automáticamente se busquen las dependencias de dicho paquete. El único problema es que el paquete principal no debe ser instalado porque ya se ha clonado y se dispone de toda la información. Una manera de evitar esto es mediante una marca en la versión. Si no se especifica una versión explícitamente, el programa podrá acceder a todos los datos porque al haber clonado el repositorio el fichero `pkg.json` estará disponible de forma local y se podrá acceder a él y poder buscar todas las dependencias. De este modo se define el convenio de definir como versión `"d.d.d"`, cuando únicamente se requieran las dependencias de un paquete. Para mantener el formato del fichero `pkg.json`, también es buena idea mantener la estructura e indicar a los paquetes dentro del campo `"dependency"`, de manera que el desarrollador no necesite conocer demasiados formatos sobre el programa. Volviendo al ejemplo del paquete A, con dependencias B y C, que necesitaba D y E para desarrollarse, se presenta una comparativa de los dos tipos de formato en la tabla 2.

Del nuevo formato, los campos `"data"` y `"exec"`, en principio quedarán vacíos y pueden ser eliminados. El motivo de dejarlos es simplemente por preservar el formato de los metadatos del paquete. Sin embargo, si no aparecen dichos campos, el programa funcionará puesto que no son necesarios. Aunque en el ejemplo anterior, no se aprecie demasiado la reducción de los datos del fichero, si un paquete tiene muchas dependencias, el hecho de poder agruparlas en solo una utilizando el `"d.d.d"` puede resultar bastante ventajoso.

3.4. Datos sobre la instalación

El último de los ficheros necesarios para el funcionamiento del programa es un fichero que almacene la información de qué paquetes están instalados, para que así se puedan desinstalar, se pueda dar un aviso cuando se intente instalar de nuevo un paquete que ya está instalado... La parte fundamental de la información que debe guardar es el nombre y la versión, porque con ello ya se puede realizar todo el control de qué está instalado y qué no. Sin embargo, ya que se va a disponer de un fichero con datos sobre la instalación, es interesante incluir nuevos campos que den mayor información al usuario que la precise. Entre esta información se encuentran los ficheros que se han copiado durante la instalación, la fecha de instalación o el tipo de paquete.

Para considerar el tipo de paquete, se ha considerado apropiada la distinción entre los paquetes necesarios en una instalación (los que aparecerían explícitamente en el `myum_install_def`, por ejemplo), de aquellos que se instalarían como consecuencias de los mismos. Esta distinción puede ser útil cuando se desinstale un paquete, para poder eliminar aquellas dependencias que ya no tienen ningún paquete del que dependen porque ya ha sido eliminado y para que el usuario conozca cuáles son los paquetes que realmente ha instalado porque los haya pedido y cuáles se han instalado como consecuencia de la instalación de otros. En esta distinción se ha establecido el valor `"requested"` para los paquetes que se instalan a través de un comando en el que se solicitan explícitamente (ya sea especificando el paquete directamente en el comando o mediante un fichero de instalación), y el valor `"dependence"` para las dependencias de los paquetes principales de la instalación.

Otro campo que resulta interesante es la fecha y hora de instalación, para tener más

Formato anterior

```
{
    "json_database_version" : "1.0",
    "packages" : [
        {
            "name"    : "B",
            "version" : "v1.0"
        },
        {
            "name"    : "C",
            "version" : "v1.0"
        },
        {
            "name"    : "D",
            "version" : "v1.0"
        },
        {
            "name"    : "E",
            "version" : "v1.0"
        }

    ]
}
```

Formato nuevo

```
{
    "json_database_version": "2.0.0",
    "packages": {
        "data": [],
        "dependency":
                        [{
                        "name": "D",
                        "version" : "1.0.0"
                        },
                        {
                        "name": "E",
                        "version" : "1.0.0"
                        },
                        {
                        "name": "A",
                        "version" : "d.d.d"
                        }
                        ],

        "exec": []
    }
}
```

Tabla 2: Comparativa de los dos formatos de `myum_install_def.json`.

datos sobre las operaciones realizadas, así como un historial de operaciones. En este historial se pueden mostrar los mismos campos de los datos de instalación, pero se pueden registrar más operaciones como una actualización, desinstalación o reinstalación que un registro de únicamente paquetes instalados no podría mantener (si se desinstala un paquete simplemente no aparecerá en la lista de instalados, por lo que se necesita un historial para contemplar la operación).

De este modo los campos básicos del fichero serían:

Para el historial:

- Nombre.
- Versión.
- Fecha y hora: Muestra la fecha de instalación, junto con la hora con precisión de segundos.
- Tipo de operación: Puede ser instalación, desinstalación, actualización o reinstalación.
- Tipo de paquete: Solicitado o dependencia.
- Ficheros instalados: Solo para el caso de instalación, reinstalación o actualización.
- Versión previa: Solo para el caso de actualización.

Para el registro de paquetes instalados:

- Nombre.
- Versión.
- Fecha y hora.
- Ficheros instalados.
- Tipo de paquete.

En la figura 15 se muestra una versión reducida de un fichero de datos de instalación (`.myum.db`). En principio, el nombre del fichero iba a ser llamado `myum_install_log.json`, pero dado que la palabra *log* puede conllevar la connotación de que es simplemente un registro y puede ser eliminado, para quedar claro que este fichero es importante y debe preservarse se ha optado por un nombre de fichero `.myum.db`, ya que el hecho de empezar por `'.'` y tener la extensión `.db` de base de datos, indica de forma más clara la necesidad de no modificar el fichero.

```
{
    "history": [{
            "name": "basic",
            "url": [
                    "design/devices/source/crip_basic/vsim/modelsim.ini",
                    "design/devices/source/crip_basic/vsim/vcompiler.bat",
            ],
            "version": "1.0.1",
            "time": "2015-04-02 21:41:57",
            "operation": "install",
            "type": "requested"
        },
        {
            "name": "arithmetic",
            "url": [
                    "design/devices/source/crip_arithmetic/vsim/vcompiler.bat",
                    "design/devices/source/crip_arithmetic/vsim/vcompiler.sh",
            ],
            "version": "1.0.0",
            "time": "2015-04-02 21:41:58",
            "operation": "install",
            "type": "command"
        },
        {
            "name": "basic",
           "url": [
                   "design/devices/source/crip_basic/vsim/modelsim.ini",
                   "design/devices/source/crip_basic/vsim/vcompiler.bat",
            ],
            "previous_version:": "1.0.1",
            "version": "2.0.0",
            "time": "2015-04-02 21:50:31",
            "operation": "update",
            "type": "command"
        },
        {
            "operation": "uninstall",
            "version": "2.0.0",
            "type": "command",
            "name": "basic",
            "time": "2015-04-02 21:52:17"
    }],
    "installed": [
        {
            "url": [
                    "design/devices/source/crip_arithmetic/vsim/vcompiler.bat",
                    "design/devices/source/crip_arithmetic/vsim/vcompiler.sh",
            ],
            "version": "1.0.0",
            "type": "command",
            "name": "arithmetic",
            "time": "2015-04-02 21:41:58"
        }
    ]
}
```


Figura 15: Formato del fichero `.myum.db`.

4. Primeras funcionalidades del programa

El siguiente paso una vez definidos los metadatos que se van a utilizar, será definir e implementar cada una de las funcionalidades que va a hacer el programa. Aunque la aplicación se desarrolle en línea de comandos y de forma gráfica, puesto que la interfaz es únicamente un *front-end* de la aplicación, el estudio se centrará en la funcionalidad general y se mostrarán ejemplos para la línea de comandos. El uso de la interfaz gráfica se dejará para un capítulo posterior.

Para la descripción de estas funcionalidades se procederá a la descripción según el orden lógico para llevar a cabo la funcionalidad de instalación y posteriormente se tratarán otras opciones complementarias que utiliza el programa. En este capítulo se analizarán solo las primeras funcionalidades y el resto se irán desengranando en capítulos sucesivos.

4.1. Ayuda del programa

La primera de las opciones que implementa el programa es la opción de ayuda, que muestra las distintas funcionalidades que presenta y la manera de activar cada una de ellas. Mediante la línea de comandos, se ha especificado que esta ayuda se muestre tanto si se ejecuta el programa sin ningún parámetro o si se utiliza la opción `-h` (o `--help`). La figura 16 muestra las distintas opciones del programa al usar al pedir la ayuda. Las funcionalidades que ofrece el `myum` son las siguientes:

- Ayuda: Muestra el mensaje de ayuda y termina la ejecución del programa.
- Versión: Muestra la versión del programa.
- Listar: Muestra todos los repositorios y paquetes disponibles.
- Selección del fichero de fuentes: Permite cambiar el fichero de fuentes por defecto por otro cuya localización especifique el usuario.
- Instalar: Instala los paquetes de acuerdo con la información del fichero de instalación.
- Selección del fichero de instalación: Permite cambiar el fichero de instalación por defecto por otro cuya localización especifique el usuario.
- Comprobación de integridad: Permite comprobar si se han producido cambios en un paquete desde la instalación o si se mantiene igual.
- Forzar: Permite forzar una instalación.
- Mostrar paquetes instalados: Muestra una lista con los paquetes instalados.
- Instalación específica: Permite instalar un paquete concreto en una versión específica utilizando directamente la línea de comandos.
- No automático: Desactiva el modo automático de resolución de dependencias.

```
$ python3 myum.py -h
usage: myum [-h] [-V] [-lp] [-m MIRRORS_FILE] [-i] [-d INSTALL_DEF_FILE]
            [-ck PACKAGE_TO_CHECK] [-f] [-u UNINSTALL_PACKAGE] [-li]
            [-s INSTALL_SPECIFIC_PKG] [-na] [-cl] [-nv] [-up] [-ar] [-hs]
            [-db] [-dr]

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -lp, --list_packages  List all the available repositories and packages
  -m MIRRORS_FILE, --mirrors MIRRORS_FILE
                        Change file containing the repositories mirrors list.
                        Default: /home/eda/myum/myum_mirror_list.json
  -i, --install         Install packets according to your install definition
                        file
  -d INSTALL_DEF_FILE, --definition INSTALL_DEF_FILE
                        use a specific install definition file instead of the
                        default (myum_install_def.json).
  -ck PACKAGE_TO_CHECK, --check PACKAGE_TO_CHECK
                        Check if the package installed has changed after
                        installation or if it remains the same.
  -f, --force           Force to install a package.
  -u UNINSTALL_PACKAGE, --uninstall UNINSTALL_PACKAGE
                        Uninstall a package.
  -li, --list_installed
                        Shows a list with the packages installed.
  -s INSTALL_SPECIFIC_PKG, --specific INSTALL_SPECIFIC_PKG
                        install a specified package given in the command line,
                        and ignore install definition file. Use format
                        pkg_a.b.c.
  -na, --noautomatic    not resolve dependencies automatically (myum will ask
                        you in case of conflicts).
  -cl, --clean          Clean cache directory.
  -nv, --noverbose      not print detail information of what is done.
  -up, --update         update all packages installed to its highest
                        compatible version.
  -ar, --autoremove     removes packages that were installed automatically to
                        satisfy dependencies from other packages and that now
                        they are not necessary
  -hs, --history        print history information contained in
                        myum_install_log (.myum.db)
  -db, --debug          enable debug mode
  -dr, --dry            simulate an myum operation without making changes
```

Figura 16: Opción de ayuda del programa.

- Limpiar: Borra la información del directorio caché.
- Modo no verboso: Desactiva el modo que muestra información detallada al usuario de las operaciones que se realizan.
- Actualizar: Actualiza todos los paquetes a la versión más alta compatible.
- Autoborrado: Elimina aquellos paquetes que se instalaron como dependencias y que ya no son necesarios.
- Historial: Imprime el historial de instalación.
- Modo depuración: Activa el modo depuración (muestra la salida de ejecución de comandos).
- Modo simulación: Activa el modo simulación para ver cuál sería el resultado de realizar ciertas operaciones sin hacer de forma efectiva los cambios.

4.2. Versión

La opción referente a la versión es muy sencilla y únicamente muestra en qué versión se encuentra el programa. Este valor se ve modificado cada vez que se haga una nueva actualización que corrija algún pequeño caso de error que se descubra o cuando se encuentre una nueva necesidad para el programa y se deban realizar los cambios oportunos para su incorporación. El formato utilizado para mostrar la versión es `myum version a.b.c` (siendo a.b.c la versión utilizando el versionado semántico descrito inicialmente; ej. 1.0.0).

4.3. Listado de paquetes

La función de listar paquetes es la primera que se plantea para el uso del `myum`. Aunque para una instalación propiamente dicha no sea necesaria, el usuario debe conocer al inicio qué es lo que está disponible para instalar.

La función principalmente hará uso del fichero de fuentes en el que se especifican los repositorios de donde tomar la información y la ubicación de los mismos. Con esta información se podrán hacer distintas llamadas al comando `git ls-remote -tags`, que sirve para recuperar la lista de `tags` de un determinado repositorio [1].

La lista de *tags* obtenida puede contener etiquetas con el formato descrito de `mtag_nombre_version`, o también puede encontrar otras etiquetas que un programador haya hecho para establecer puntos en su programa o incluso los *tags* existentes antes del desarrollo de la herramienta. Por ello, es necesario realizar una limpieza y quedarse con únicamente aquellas etiquetas adecuadas y luego procesarlas para obtener el nombre y versión, para obtener una lista.

Para empezar la labor de obtener etiquetas se presentaban dos alternativas de uso. En primer lugar, lo más sencillo es tomar cada uno de los repositorios, obtener sus *tags*, procesarlos y almacenar el resultado en una lista para que el usuario pueda ver el contenido disponible.

Entre las ventajas de esta alternativa, destaca su alta sencillez, puesto que lo único que implica es realizar llamadas a un proceso de forma secuencial (y su posterior procesamiento, pero que esto es inevitable de todas formas). Sin embargo, cuenta con la gran desventaja de tener que esperar a obtener el resultado de un comando para poder ejecutar el siguiente, cuando al ser comandos independientes con resultados independientes, se podría paralelizar el proceso.

Precisamente, la segunda alternativa es utilizar distintos hilos de ejecución para cada uno de las llamadas a *Git* y recoger finalmente todos los resultados. Como el número de repositorios puede crecer indefinidamente, este hecho mejora mucho la escalabilidad y la velocidad de procesamiento. Como inconvenientes, el principal es la mayor complejidad y los problemas que pueden ocurrir si ocurre una excepción (porque el servidor esté caído, no existan las credenciales para acceder a *Git*, etc.) al ejecutar el programa. Sin embargo, se considera que el uso adecuado puede resultar bastante ventajoso. La figura 17 muestra el proceso seguido para la obtención de la lista de paquetes.

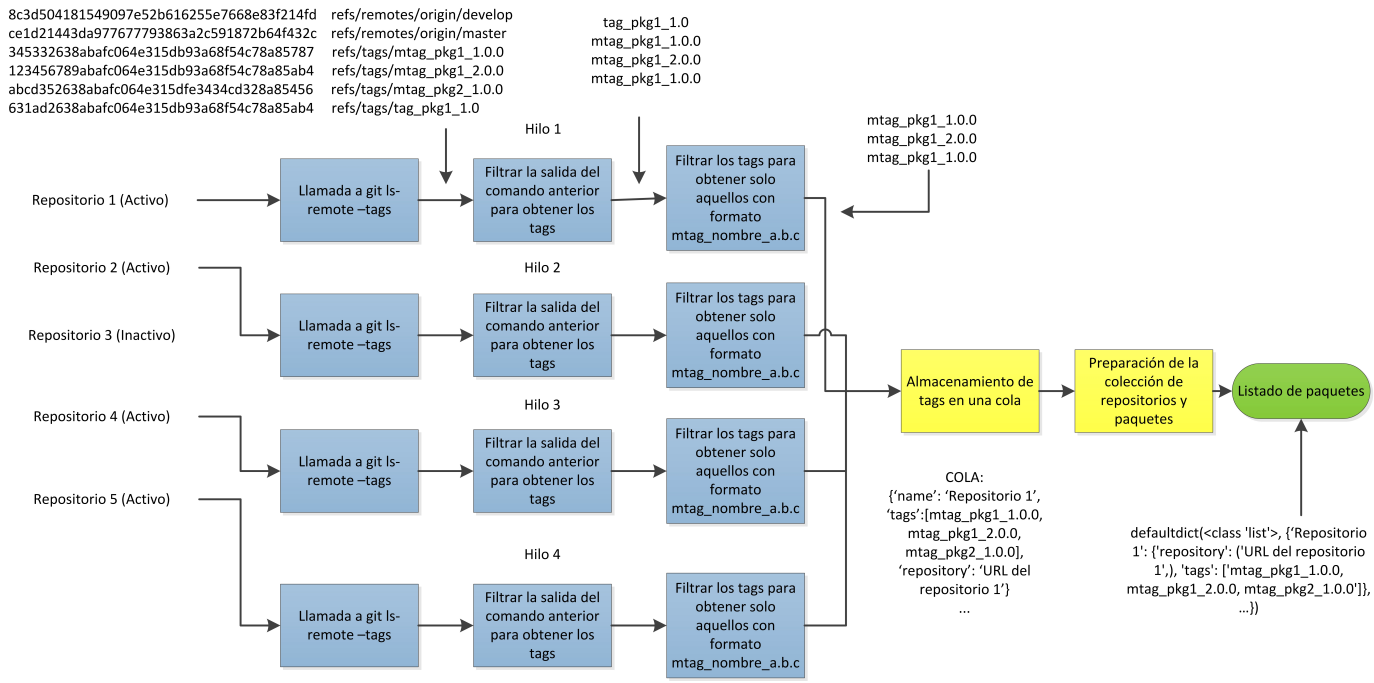

Figura 17: Esquema del procedimiento de obtención de la lista de paquetes.

Con la estructuras de hilos presentada, se tomará cada uno de los repositorios activos y en un hilo de ejecución se hará la llamada a *Git* para obtener los *tags*. Sin embargo, la salida no proporciona directamente el resultado esperado, sino que muestra todas las referencias de los *tags*(incluyendo el punto más reciente donde se sitúan las distintas ramas de trabajo dentro del repositorio). El aspecto que presenta esta salida se muestra en la figura 18.

Ante esta salida, lo que se aplicará es una expresión regular (secuencia de búsqueda utilizada para encontrar los patrones del formato de *tag* dentro de la salida inicial) [14], que permita quedarse solo con el nombre de los *tags*, que están dentro de una referencia del tipo `refs/tags/`, eliminando de este modo los puntos de las distintas ramas.

Una vez obtenidos los *tags* de un repositorio en concreto, puede ocurrir que haya

```
f67e297c95efd9f2243dd7df3fdafa648742a672        HEAD
3adc155950200b85f9ae7ce30c68a6ab19f109be        refs/heads/develop
f67e297c95efd9f2243dd7df3fdafa648742a672        refs/heads/feature/migration
fe3f85b1015086b590d6af5b15feb98a03733234        refs/heads/master
0796f3e564679bb48ef02b0bc22c0b4a7138bc47        refs/heads/myum_dependency
ce1d21443da977677793863a2c591872b64f432c        refs/remotes/origin/HEAD
8c3d504181549097e52b616255e7668e83f214fd        refs/remotes/origin/develop
ce1d21443da977677793863a2c591872b64f432c        refs/remotes/origin/master
631ad2638abafc064e315db93a68f54c78a85ab4        refs/tags/mtag_pkg1_1.0.0
5435638abafc064e315db45368f54c78a8878788        refs/tags/mtag_pkg1_1.0.2
ab354fe3578c064e315db45368f54c78a8878788        refs/tags/mtag_pkg2_1.0.0
9877665421ab064e315db45368f54c78a8874446        refs/tags/tag_pkg_1.0.0
```

Figura 18: Salida de la función `git ls-remote -tags`.

tags que no estén en el formato deseado. Para cubrir esto, se busca qué *tags* tienen la marca de `mtag`. Con éstos, se elabora un diccionario en el que aparecerá la relación de nombre del repositorio, etiquetas y su *URL*, que es lo necesario para que a partir de un paquete se le pueda encontrar en su lugar correspondiente. Estos datos se guardan en una cola que irá almacenando el resultado de cada uno de los hilos.

Uno de los principales problemas de compatibilidad entre versiones de *Python* ocurren en situaciones como esta y es que mientras que el módulo que implementa las colas en *Python 3* se llama *queue*, el que lo hace en *Python 2* se llama *Queue*, con la primera letra en mayúscula [15]. Para solucionar este problema, se debe comprobar al inicio la versión con la que se ejecuta el programa e importar el módulo con el nombre adecuado según cambie la versión.

Otro de los problemas de la cola es que si ocurriese una excepción durante la ejecución de los hilos (por ejemplo, porque haya un problema con la conexión al servidor), el programa principal se quedaría congelado, puesto que se queda a la espera de recoger el resultado de todos los hilos y nunca podría recoger el resultado de aquel que levantó la excepción. Para ello, la solución que se ha llevado a cabo es que si un hilo produce una excepción, que se capture la misma y se inserte una lista de *tags* vacía en la cola para que la ejecución pueda continuar y se puedan mostrar los repositorios donde no haya habido problemas, aparte de mostrar la advertencia sobre los problemas del repositorio.

Con los datos recogidos de la cola por cada uno de los hilos, se construye una colección de diccionarios que almacenen todos los datos recogidos.

Por último, con la estructura preparada, si el usuario ejecuta la opción `-lp` (o `--list_packages`), se le listará la lista de paquetes (para el resto de opciones, la colección se creará para ser utilizada posteriormente). El único factor que se deberá tener en cuenta es que los paquetes están almacenados en la colección de forma desordenada, por lo que para que el usuario pueda buscar los paquetes que quiere fácilmente, se lleva a cabo un orden alfabético, de modo que los paquetes aparezcan de forma alfabética y por cada paquete aparezcan listadas cada versión también ordenada de la más reciente a la más actual. Para marcar además cuál es la versión más alta compatible, estas versiones aparecerán con un `*` y con la línea de color verde (excepto en *Windows*, que dado que no permite utilizar secuencias de escape [16], se ha optado por mantener los mantener las letras en blanco). La figura 19 muestra un ejemplo de salida de la opción `-lp`.

En la tabla que se muestra en la salida, aparece en la primera columna el nombre del paquete, en la segunda la versión, en la tercera el nombre del repositorio donde encuentra y en la última la dirección de dicho repositorio. Adicionalmente, aparece el directorio de trabajo y el fichero de fuentes utilizado al inicio para que el usuario sepa desde donde está funcionando el programa y a partir de qué se ha elaborado la lista de paquetes.

Según lo mostrado en la salida, se puede observar cómo un mismo paquete aparece marcado con el `*` varias veces (ej: *apb*). El motivo es que por cada versión `X.Y.Z` con `X` constante, aquella con mayor `Y.Z`, será la mayor compatible pero para ese `X`. Si este varía, habrá otra versión más alta compatible para el nuevo `X`. En el ejemplo, el paquete existe en versión 1.0.1, 1.1.0, 2.0.0 y 2.1.0. Dentro de las versiones 1, la más alta es la 1.1.0 y por ello está marcada; y dentro de las versiones 2, la que se marca es la 2.1.0.

```
myum INFO: Workspace is /cygdrive/d/myum/pmoreno
myum INFO: Repository mirror list file is mirror_list_cygwin.json
myum INFO: These are the available packages:
agent_adc                       1.0.0   crip_adc          /cygdrive/d/myum/projects/crip_adc/.git
agent_adc                       1.0.1   crip_adc          /cygdrive/d/myum/projects/crip_adc/.git
agent_adc                       1.1.0 * crip_adc          /cygdrive/d/myum/projects/crip_adc/.git
agent_apb                       1.0.0   crip_apb          /cygdrive/d/myum/projects/crip_apb/.git
agent_apb                       1.0.1 * crip_apb          /cygdrive/d/myum/projects/crip_apb/.git
agent_clkrst                    1.0.0 * crip_setup        /cygdrive/d/myum/projects/crip_setup/.git
agent_common                    1.0.0 * crip_setup        /cygdrive/d/myum/projects/crip_setup/.git
agent_eeprom_3dee8w08vs8190     1.0.0 * crip_memory       /cygdrive/d/myum/projects/crip_memory/.git
agent_eeprom_hm58v1001          1.0.0 * crip_memory       /cygdrive/d/myum/projects/crip_memory/.git
agent_extmem                    1.0.0   crip_memory       /cygdrive/d/myum/projects/crip_memory/.git
agent_extmem                    1.0.1 * crip_memory       /cygdrive/d/myum/projects/crip_memory/.git
agent_real                      1.0.0 * crip_setup        /cygdrive/d/myum/projects/crip_setup/.git
apb                             1.0.0   crip_apb          /cygdrive/d/myum/projects/crip_apb/.git
apb                             1.1.0 * crip_apb          /cygdrive/d/myum/projects/crip_apb/.git
apb                             2.0.0   crip_apb          /cygdrive/d/myum/projects/crip_apb/.git
apb                             2.1.0 * crip_apb          /cygdrive/d/myum/projects/crip_apb/.git
arithmetic                      1.0.0   crip_arithmetic   /cygdrive/d/myum/projects/crip_arithmetic/.git
arithmetic                      1.1.0 * crip_arithmetic   /cygdrive/d/myum/projects/crip_arithmetic/.git
arithmetic                      2.0.0 * crip_arithmetic   /cygdrive/d/myum/projects/crip_arithmetic/.git
basic                           1.0.0   crip_basic        /cygdrive/d/myum/projects/crip_basic/.git
basic                           1.0.1 * crip_basic        /cygdrive/d/myum/projects/crip_basic/.git
basic                           2.0.0 * crip_basic        /cygdrive/d/myum/projects/crip_basic/.git
common                          1.0.0 * crip_setup        /cygdrive/d/myum/projects/crip_setup/.git
crlogger                        1.0.0 * crip_setup        /cygdrive/d/myum/projects/crip_setup/.git
driver_common                   1.0.0 * crip_setup        /cygdrive/d/myum/projects/crip_setup/.git
memory                          1.0.0 * crip_memory       /cygdrive/d/myum/projects/crip_memory/.git
technology                      1.0.0 * crip_technology   /cygdrive/d/myum/projects/crip_technology/.git
```

Figura 19: Salida por pantalla tras listar la lista de paquetes.

4.4. Selección del fichero de fuentes

Anteriormente, se ha expuesto la necesidad del fichero de fuentes y el uso práctico dentro del programa para obtener la lista de paquetes (y mostrarla si procede). Dado que este fichero estándar de los repositorios es bastante estático, puesto que no con mucha frecuencia aparecen repositorios nuevos, sino que aparecen paquetes dentro de los repositorios actuales y si esto ocurre al buscar en el repositorio, los nuevos paquetes serán encontrados; entonces se ha decidido mantener un fichero de fuentes común para que por defecto el programa se alimente de él. Este fichero contendrá todos los repositorios disponibles en el servidor de *Git* de la sección, *VMGitLab*, y tendrá distinta localización dependiendo del sistema operativo, por lo que será necesario comprobar el sistema operativo al inicio para utilizar la localización del fichero adecuada.

En *Windows*, el fichero se localiza en la máquina `fs2` dentro del directorio `X:\myum\myum_mirror_list.json`. En cambio, en *Linux*, se sitúa dentro de `/home/eda/myum/myum_mirror_list.json`. Por último tenemos el caso de *Cygwin*, que es un entorno Linux para Windows, consistente en un *DLL* (`cygwin1.dll`), que actúa como una capa de emulación que proporcionan la funcionalidad de las llamadas del sistema *POSIX* (*Portable Operating System Interface)* y un conjunto de herramientas, que proporcionan el aspecto y apariencia de *Linux*. En este caso se utiliza la dirección utilizada en *Linux*, dado que aunque por defecto *Cygwin* utiliza sus propias direcciones, como normalmente están mapeadas éstas con las de *Linux*, se ha optado por utilizar dicha dirección.

No obstante, cualquiera que desarrollase un paquete, podría tener el interés de hacer pruebas e instalarle y desinstalarse sin que todavía el paquete estuviese disponible para el uso global de la sección. Para ello, se podría crear un fichero de fuentes propio en el cual se especificase un repositorio local donde se encontrase dicho directorio de pruebas. Para decirle al programa que se utilice un fichero determinado en lugar del descrito antes por defecto se utiliza la opción `-m` (o `--mirrors`) junto con el nombre del fichero (ej. `myum.py -m mi_fichero_de_fuentes.json`, tomaría el fichero `mi_fichero_de_fuentes.json` dentro del directorio donde se ejecute el programa; o `myum.py -m /home/fichero.json` tomaría el fichero `fichero.json` utilizando la ruta absoluta dada).

4.5. Selección del fichero de instalación

Al igual que se ha descrito en el subapartado anterior, también es posible definir un fichero de instalación propio. Por defecto se utilizará el `myum_install_def.json`, que se encuentra dentro del mismo directorio desde donde se invoca el programa. Sin embargo, es posible que un usuario realice una instalación y se quiera guardar la configuración de lo que ha instalado y copiar el fichero de instalación y usar otro con distinto nombre, de modo que si no le gustasen los cambios, desinstalase todo y finalmente pidiese una instalación según el otro fichero que se guardó. Para esos casos, puede ser útil poder definir otro fichero de instalación; o también para el caso en el que se ejecute el programa desde otro directorio por alguna circunstancia.

Para poder definir el fichero de instalación, en línea de comandos existe la opción `-d` (o `--definition`), a la que se acompaña el nombre del fichero de instalación, que se puede especificar de nuevo mediante una dirección relativa o absoluta.

5. Instalación de paquetes

La siguiente funcionalidad que se describirá es la más importante y la que forma el núcleo del programa, de ahí que se dedique un capítulo completo: la instalación. Instalar un paquete consiste en obtener una serie de ficheros de un repositorio y depositarlos en un lugar determinado. Sin embargo, aunque la tarea parece sencilla, requiere una larga serie de pasos para, en general, llevarse a cabo con éxito, que se describen a continuación:

1. Comprobar que el repositorio está limpio, es decir que no haya cambios por *commitear*. [1]
2. Elaborar la colección con todos los paquetes disponibles siguiendo el proceso explicado con anterioridad.
3. Obtener la lista de paquetes que se quieren instalar descritos en el fichero de instalación.
4. Obtener y resolver las dependencias de los distintos paquetes (este es el proceso más complejo).
5. Analizar si un paquete debe ser instalado o no de acuerdo a los registros de instalación (si un paquete está ya instalado puede ser que no deba volver a instalarse de nuevo).
6. Si un paquete instalado debe volver a instalarse, comprobar su integridad.
7. Desinstalar los paquetes que sean necesarios para poder instalar otra versión en los mismos (o la misma en caso de reinstalar).
8. Descargar el contenido de los paquetes de los repositorios y depositarlos en el lugar donde se realice la instalación.
9. Establecer los permisos de lectura, escritura y ejecución adecuados para los ficheros instalados.
10. Ejecutar los comandos adecuados para completar la instalación (normalmente *scripts* de post-instalación)
11. Actualizar el registro de paquetes instalados y el histórico de instalación.
12. Actualizar el fichero de instalación si procede con la configuración de instalación actual.

De un primer vistazo, parece un proceso bastante complejo. En las próximas subsecciones se irá desengranando cada uno de los distintos pasos para comprender mejor el proceso de instalación.

5.1. Comprobación del estado del repositorio

El primer paso que `myum` realizará antes de comenzar cualquier instalación es comprobar el estado del repositorio donde se esté trabajando. Con el fin de que si alguien quiere

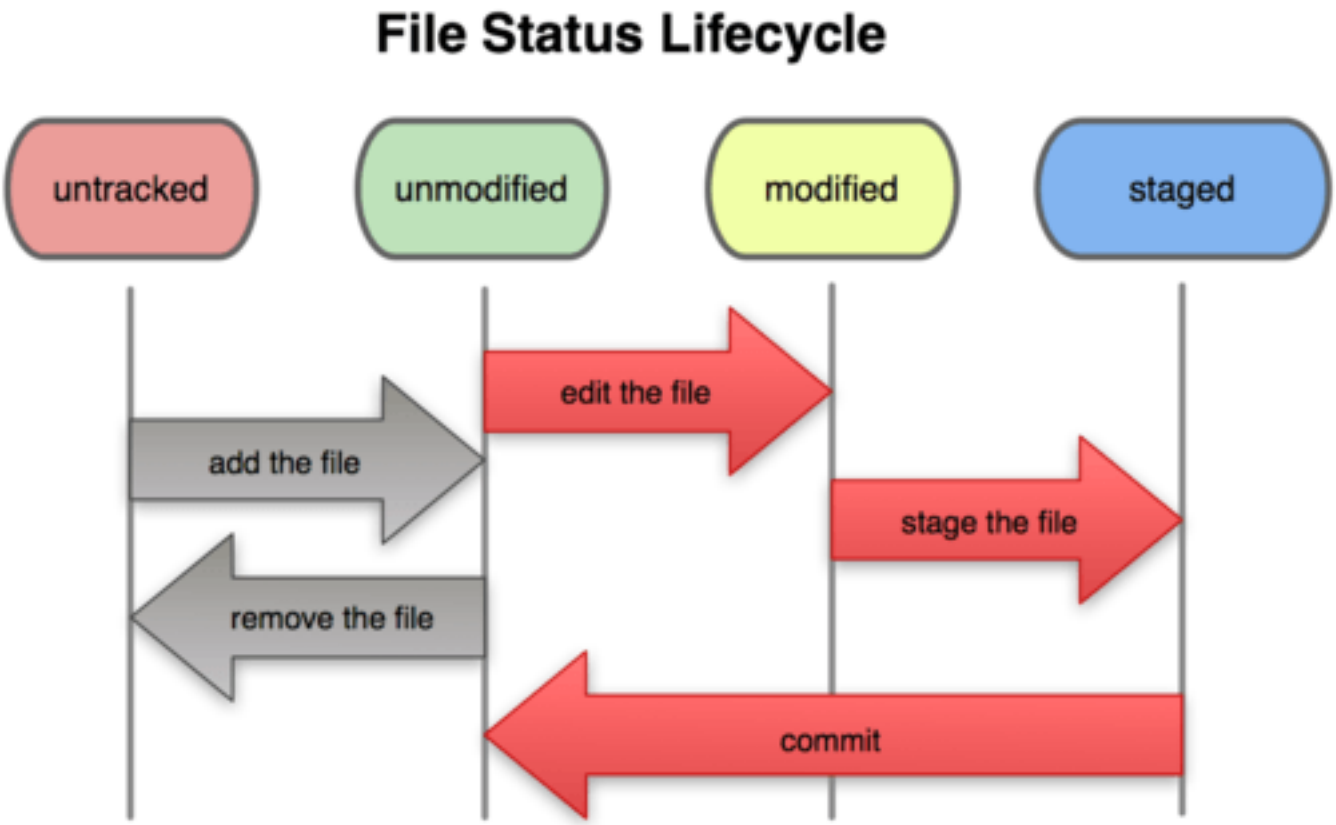

Figura 20: Ciclo de vida del estado de los ficheros. Fuente: *Git*. [1]

realizar cualquier instalación, que se asegure de que los cambios estén registrados por si se necesita posteriormente volver hacia atrás, se obliga al usuario a que haya guardado todos los cambios y el repositorio se encuentre totalmente limpio.

Para entender mejor este concepto, es necesario precisar qué se entiende por repositorio limpio. Cuando se tiene un repositorio *Git*, pueden existir muchos ficheros, pero algunos de ellos pueden estar bajo control de versiones y otros no. Aquellos que no están bajo control de versiones se les denomina *untracked*. Para empezar, si un repositorio contiene ficheros de este tipo, se considerará que no está limpio. Estos ficheros deben ser añadidos y confirmados (mediante un *commit*) para que empiecen a funcionar bajo el control de versiones. Para entender mejor la explicación, la figura 20 muestra un ejemplo del ciclo de vida del estado de los ficheros en *Git*.

En el caso anterior, pasamos de tener un fichero *untracked* a un fichero *staged* (al añadir un fichero por primera vez, el comando de añadir (`git add`) también prepara el fichero para su confirmación. Una vez que el fichero está confirmado mediante el *commit*, pasa al estado *unmodified*. Si todos los ficheros estuvieran en este estado, entonces el repositorio se consideraría limpio.

También puede ocurrir que un fichero bajo control de versiones se esté desarrollando y se modifique. En ese caso, pasaría a estar en estado *modified*. Teniendo ficheros en este estado, el repositorio se consideraría no limpio y la instalación no se llevaría a cabo. Para poder continuar,

habría que preparar los cambios para la próxima confirmación (lo que se llama hacer un *stage*) y finalmente confirmar los cambios. para pasar al estado *unmodified* de nuevo.

Para comprobar el estado de un repositorio, *myum* hará uso del comando `git status`, que te indica si el repositorio está limpio o no, y en caso negativo qué ficheros hacen que no lo esté y en qué situación se encuentran (si no están bajo control de versiones, si están añadidos pero no confirmados, si están modificados, si están eliminados, renombrados, copiados o actualizados). Al comando se le añade la opción `--porcelain` para poder ser procesada la salida fácilmente.

Uno de los problemas que genera esto es que el propio *myum* escribe y modifica archivos durante su ejecución (el *log* de operaciones, el fichero de instalación y el fichero de datos sobre la instalación). Si aplicamos el comando y miramos únicamente la salida, nunca se puede conseguir un directorio limpio, pues aun cuando todo esté confirmado, nada más arrancar el programa se registrará algún mensaje de notificación que harán que el registro de mensajes cambie y el repositorio ya no esté limpio.

Para solucionar este problema, en el procesado de la salida del comando se ha añadido un procesado adicional que permite eliminar el nombre de archivos concretos, principalmente generados por el propio programa. En este caso, el fichero de datos de instalación tiene el nombre `.myum.db` y el de los mensajes, que se describirá más adelante, se llama `myum.log`. Respecto al fichero de definición de la instalación, generalmente se llama `myum_install_def.json`, pero mediante la opción `-d`, podría ser cualquier otro con otro nombre. Por ello, se debe leer qué nombre concreto se usa para eliminar el adecuado.

Una alternativa a este procesamiento es utilizar el fichero `.gitignore`, que permite indicar qué ficheros se deben omitir. Si se utilizase este método, directamente la salida del comando no mostraría los ficheros anteriores como modificados y daría el repositorio limpio sin necesidad de un procesamiento adicional. Sin embargo, tiene el inconveniente de que para conseguir este efecto es necesario modificar el fichero. Hacer esto de forma manual no parece muy sensato, puesto que se obliga al usuario a realizar acciones que podría no saber que tener que hacer o de qué modo; pero modificar este archivo de forma automática, aunque se podría hacer, tampoco se estima como la mejor solución puesto que se estarían modificando metadatos de otro programa diferente, lo cual no es la labor del *myum* y hacerlo sería peligroso en el sentido de que estaría modificando el comportamiento de otro programa independiente. Por ello, la solución propuesta se estima como la más adecuada, aunque trabaje con salidas del comando de *Git* no limpias y las convierta.

5.1.1. Definición de la raíz del proyecto

Con respecto a lo comentado hasta ahora, parece una obligación la necesidad de trabajar dentro de un repositorio, pues si no se está dentro no tiene ningún sentido comprobar si está limpio o no. Sin embargo, el programa es capaz de definir directorios de trabajo fuera de cualquier repositorio. Para ello se utilizará la siguiente política:

- En primer lugar se utilizará la variable de entorno `WORKSPACE`. Si esta variable está

definida, se tomará directamente su valor.

- Si no existe la variable de entorno `WORKSPACE`, se entenderá por defecto que el usuario se encuentra en un repositorio y se buscará la raíz del mismo. La raíz se encuentra en el directorio donde se encuentre por debajo el directorio llamado `.git`. En el siguiente ejemplo de directorios, el directorio raíz sería el `subdirectorio1` porque bajo él se encuentra el `.git` y este resultado se obtendría tanto si se llamase el programa desde `subdirectorio1`, `subdirectorio3` o `subdirectorio4` (aparte del propio .git) porque se subiría hacia arriba los niveles que hiciesen falta hasta encontrar el directorio deseado.

```
--C:
   --directorio1
        --subdirectorio1
             --.git
             --subdirectorio3
                  --subdirectorio4
        --subdirectorio2
   --directorio2
```

- El último de los casos es el que ocurriría si estuviésemos en `directorio2` o `subdirectorio2` sin haber definido el `WORKSPACE`. En esos casos, por muchos niveles de profundidad que se subiesen al final se acabaría en el nivel superior sin encontrar el `.git` puesto que no se está dentro de un repositorio. Para dichos casos se toma como raíz del proyecto el propio directorio.

Y una vez que se tiene el directorio de trabajo, ¿qué pasa si no se está en un repositorio? En principio, aunque se haya definido un directorio, el programa dará un error, pero ahora notificará que no se está en un repositorio en vez de decir que no está limpio y no dejará continuar la instalación.

Para solucionar este problema existe la opción `-f` para forzar la instalación. Cuando esta opción se activa, las comprobaciones de los repositorios serán omitidas, si bien no se dejarán de realizar, y en caso de dar un resultado negativo, se notificará al usuario, aunque no se detendrá el proceso de instalación. El *flag* de forzado será tenido en cuenta para más casos problemáticos durante el resto de la instalación.

5.2. Obtención de la lista de paquetes disponibles

El segundo paso para realizar una instalación es obtener la colección de paquetes disponibles. Como este proceso ya se describió cuando se explicó la opción `-lp`, simplemente en este caso se utilizarían las mismas funcionalidad, con la única salvedad es que ahora ya no se debe mostrar por pantalla los paquetes disponibles.

```
myum INFO: The following packages were required to install:
1    basic              1.0.0     (requested)
2    arithmetic         1.0.0     (requested)
3    apb                Only dependencies
```

Figura 21: Listado de paquetes requeridos.

5.3. Obtención de la lista de paquetes requeridos

El siguiente paso es la obtención de la lista de paquetes que el usuario quiere que se instalen por petición propia y que están contenidos en el fichero de instalación. Para ello, se debe leer dicho fichero y extraer la lista encontrada dentro de la clave `"dependency"`.

Este proceso, aunque es bastante sencillo tiene un par de casos principales en los que puede fallar y para los cuales se ha realizado un manejo de excepciones específico para notificar al usuario del problema. Estos casos son:

- El fichero de instalación contiene algún error de sintaxis (ej. falta alguna coma entre paquete y paquete). En ese caso, *myum* devolverá un mensaje en el que aparecerá en qué línea de código se ha producido el error sintáctico y cuál es el motivo del mismo para facilitar al usuario la corrección del problema.
- El segundo de los problemas es que se especifique una versión que no esté en versionado semántico. Dado que este sistema de versiones es novedoso en la sección, podría ser que alguien utilizase el tradicional vX.Y (ej. v1.0), lo cual no sirve. En ese caso se notificaría explícitamente qué paquete tiene el formato de versión incorrecto. Se recuerda que también se acepta el formato `d.d.d` cuando solo se quieran instalar las dependencias de un paquete.

Una vez realizadas estas comprobaciones se debe tener una lista con el siguiente formato:

```
[{'name': 'pkg1', 'version': '1.0.5'},
 {'name': 'pkg2', 'version': '2.2.0'},
 {'name': 'pkg3', 'version': '1.0.0'},
 {'name': 'pkg4', 'version': 'd.d.d'}]
```

Antes de proceder al siguiente paso, se mostrará por pantalla (salvo que el usuario especifique lo contrario mediante la opción de no verbosidad) la lista de paquetes requeridos. La figura 21 muestra el mensaje que se daría cuando se piden para instalar los paquetes *basic*, *arithmetic* y se piden únicamente las dependencias del *apb*.

5.4. Resolución de dependencias

Dentro del proceso de instalación, el paso más conflictivo y el que determinará el buen o mal uso de la herramienta es el de resolución de dependencias. Como se expuso al inicio, es posible que para que un paquete funcione necesite la instalación de uno o más paquetes adicionales. Estos paquetes son llamados dependencias. El principal problema de la instalación,

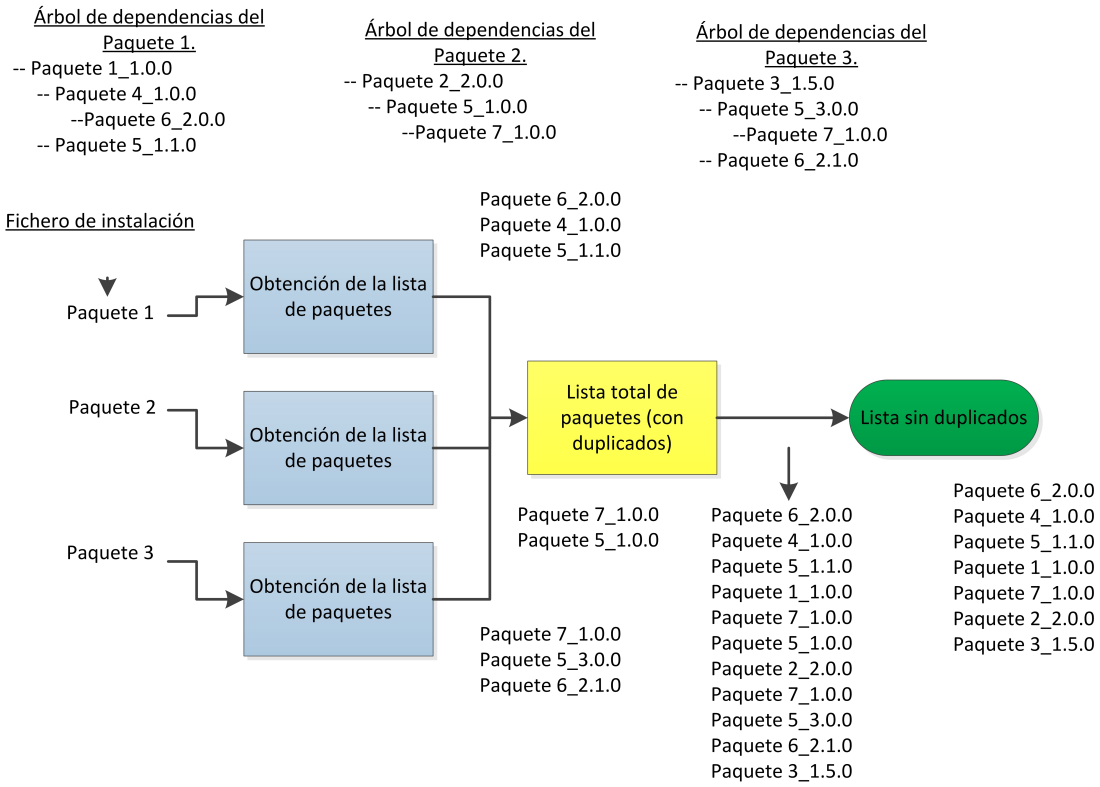

Figura 22: Resolución inicial de dependencias.

es que al listar todas éstas, podría ocurrir que un mismo paquete apareciese más de una vez. Si apareciese siempre en la misma versión, no habría problema porque se eliminarían las repeticiones, pero al aparecer en distintas versiones, es necesario diseñar un mecanismo que se encargue de determinar en qué versión se debe instalar el paquete.

Para comprender la importancia de este bloque, se analizará como primera alternativa mantener la filosofía de resolución de dependencias del antiguo programa. En esta versión, se tomaba cada paquete del fichero de instalación y se buscaban las dependencias recursivamente. La figura 22 muestra de forma más clara este proceso.

Según el esquema, en el fichero de instalación aparecerían los paquetes 1, 2 y 3. En primer lugar, se buscarían las dependencias para el paquete 1, que serían el paquete 4 en versión 2.0.0 (que a su vez tendría como dependencia el paquete 6) y el paquete 5 en versión 1.1.0; después se haría lo propio con el paquete 2 y con el 3. Tras el primer procesado se obtendrían solo las dependencias y posteriormente una lista larga con los paquetes y las dependencias según el orden de obtención que viene dado por el orden del fichero de instalación.

En dicha lista es posible que aparezca un mismo paquete repetido y en distintas versiones que no sean compatibles (ej. el paquete 5 aparece en versiones 1.1.0, 1.0.0 y 3.0.0 cuando las versiones que empiezan por 1 y las que empiezan por 3 no son compatibles). En esta aproximación inicial se tomaba la primera versión que aparecía en la lista.

Esta solución, sin embargo, conlleva un montón de problemas, puesto que en el mejor de los casos de que las versiones compatibles, puede estarse escogiendo una versión más antigua;

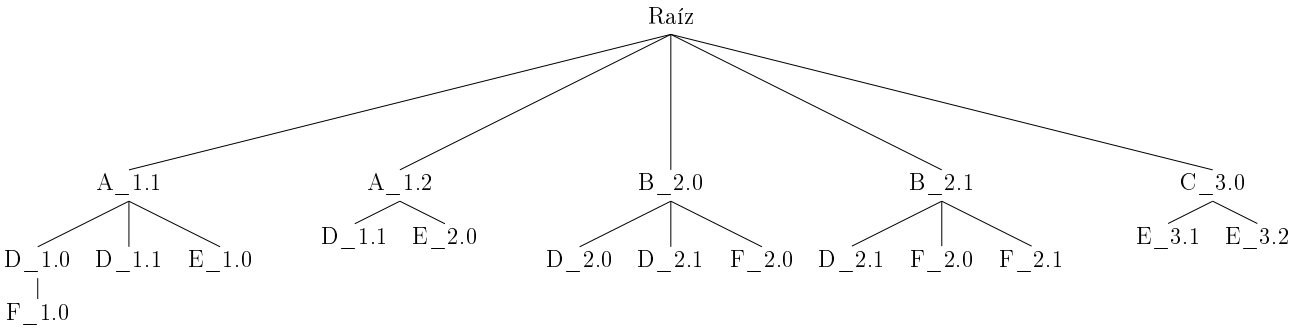

Figura 23: Árbol de dependencias inicial.

y ya si son incompatibles, puede que no se elija la más apropiada y ni se da la posibilidad de preguntar al usuario. De hecho, este sistema de resolución de conflictos fue el que dio lugar a que el programa original no diera buenos resultados.

La segunda de las aproximaciones se basa en modelizar las dependencias según un modelo matemático [17]. Para realizar este algoritmo, lo que se proponía es que en el fichero de instalación, en lugar de especificar la versión concreta en las dependencias, que se pudiera especificar la versión parcialmente, dejando con una X los parámetros irrelevantes (ej. la versión 1.X.X significa que dentro de una versión, que sea válida cualquiera que empiece por 1). Con esto, se podrían construir diferentes árboles con distintas posibilidades. Antes de continuar, se expondrá un ejemplo para proseguir con la explicación (por simplicidad se tomarán únicamente dos dígitos en las versiones). La figura 23 ilustra este ejemplo.

Se supone que en un fichero de instalación se pide instalar los paquetes A_1.X, B_2.X y C_3.X. Resulta que del paquete A existen las versiones 1.1 y 1.2; del paquete B las versiones 2.0 y 2.1; y del paquete C la versión 3.0. Las dependencias de A_1.0 son D_1.X (que podría ser D_1.0 y D_1.1, teniendo a su vez D_1.0 de dependencia F_1.0) y E_1.0; y las de A_1.1 son D_1.1 y E_2.0. Para el caso de B, B_2.0 depende de D_2.X (existiendo D_2.0 y D_2.1 sin dependencias) y F_2.0; y B_2.1 depende de D_2.1 y F_2.X (existiendo F_2.0 y F_2.1). Por último, C tiene versiones C_3.0, depende de E_3.X, que existe E_3.1 y E_3.2.

En primer lugar se analizarían las dependencias comunes de la primera rama, en este caso A. De ahí, se puede comprobar que siempre hay un D_1.X porque A_1.1 depende de D_1.0 o D_1.1; y A_1.2 depende de D_1.1. Con esto, ya podemos saber que B_2.0 o B_2.1 no son compatibles porque requieren una versión incompatible de D. Por tanto, estas ramas caerían del árbol. Como no queda ningún B activo, entonces este árbol no convergería y la instalación no se llevaría a cabo por problemas de dependencias.

Supongamos que el árbol inicial fuese más grande y quedase una versión B_2.2, con dependencias a F_2.0 y F_2.1 (figura 24). Entonces, como queda todavía una rama activa para paquetes B se continuaría el algoritmo y se pasaría al paquete B. De todos los paquetes B, lo único que hay en común es F_2.X. De este modo, aquellas ramas que tienen un F_1.X quedarían podadas. En este caso únicamente se podaría la rama A_1.1. Entonces nos quedaría el árbol de la figura 25.

Como todavía queda una rama activa para el paquete A, continuaría el proceso y se

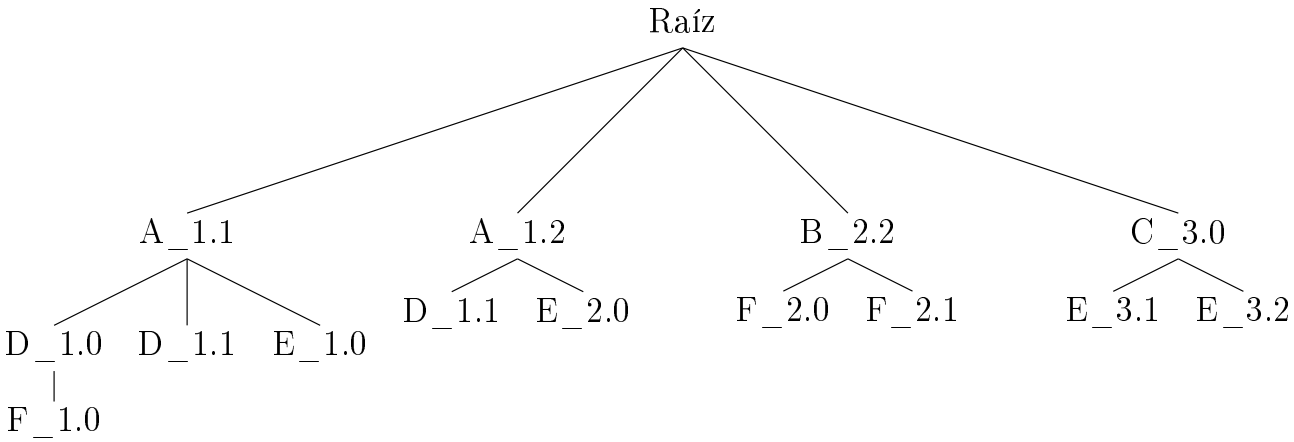

Figura 24: Árbol de dependencias tras podar B_2.0 y B_2.1.

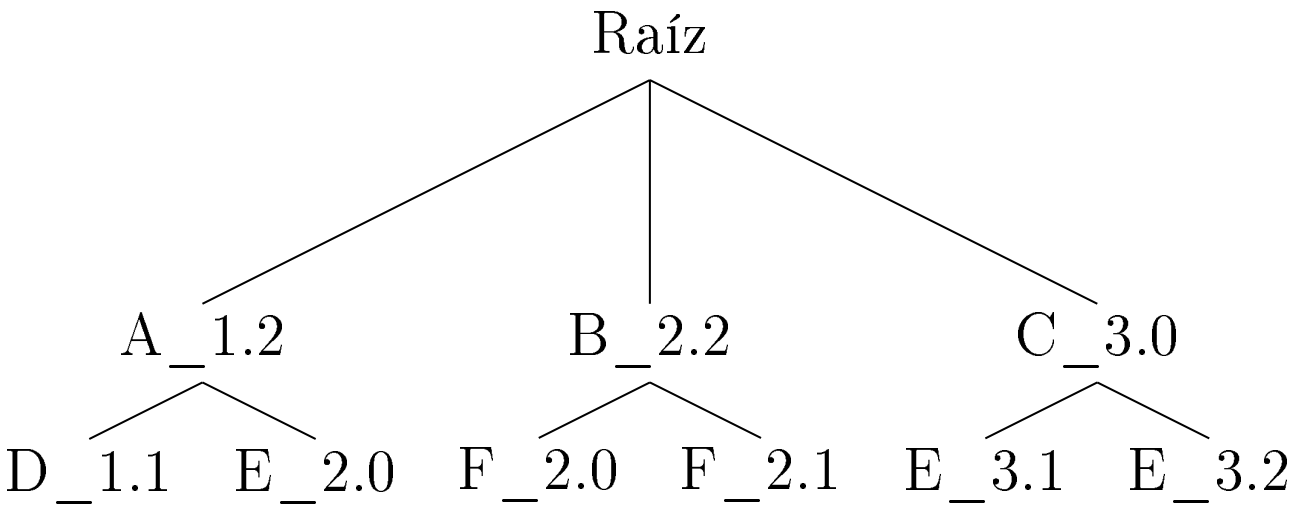

Figura 25: Árbol de dependencias tras podar A_1.1.

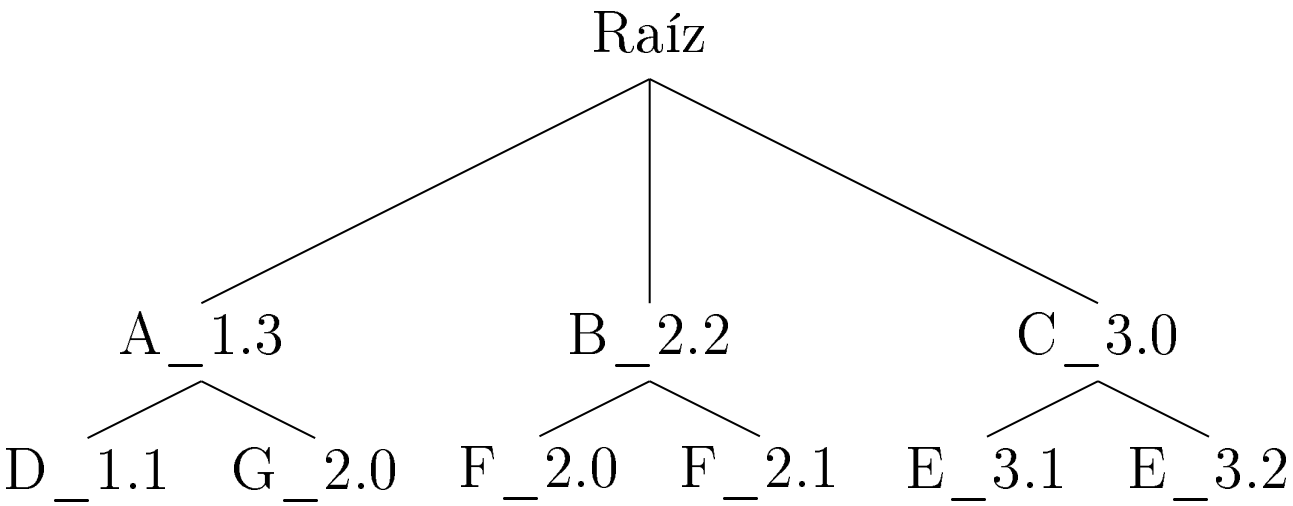

Figura 26: Árbol de dependencias convergente.

pasaría al paquete C, que daría como paquetes comunes los E_3.X, eliminaría la rama de A y finalmente el árbol no convergería. En caso de que quedase una rama activa, se volvería al paquete A repitiendo el proceso hasta que no hubiera cambios en una ronda completa.

Supongamos la aparición de una nueva versión A_1.3, que nos daría después de aplicar el proceso el árbol convergente de la figura 26 tras ejecutar el algoritmo.

Una vez que el algoritmo haya convergido, se tendrán varias posibles soluciones:

1. Instalar A_1.3, B_2.2, C_3.0, D_1.1, E_3.1, F_2.0 y G_2.0.
2. Instalar A_1.3, B_2.2, C_3.0, D_1.1, E_3.1, F_2.1 y G_2.0.
3. Instalar A_1.3, B_2.2, C_3.0, D_1.1, E_3.2, F_2.0 y G_2.0.
4. Instalar A_1.3, B_2.2, C_3.0, D_1.1, E_3.2, F_2.1 y G_2.0.

La siguiente pregunta sería qué opción de entre todas escoger. Para ello, la modelización del problema sugiere atribuir unos pesos a distintas características de cada escenario final para decidir cuál es el más apropiado. Entre estas se podría definir el tamaño del árbol, el número de paquetes totales, el número de paquetes que habría que instalar porque algunos haya instalados, etc.

Aunque se ha presentado este sistema, que mejora significativamente el inicial que no tomaba ningún criterio fiable para tomar los paquetes, también tiene una serie de inconvenientes:

- Complejidad. El algoritmo y procesamiento que se necesita es mucho más complejo.
- Fiabilidad. Para que el algoritmo sea fiable es necesario establecer los pesos adecuados para escoger el escenario final. La elección manual podría ser inadecuada o sujeta a una

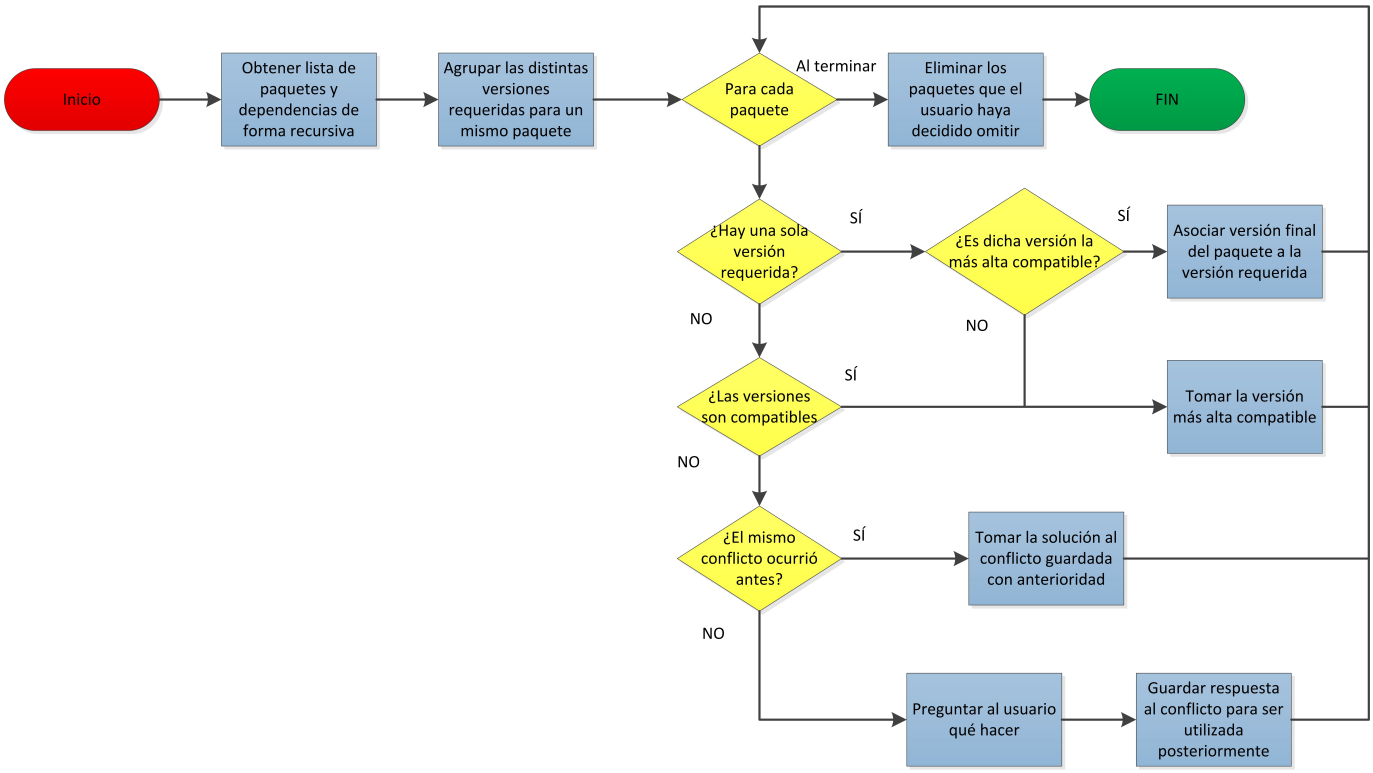

Figura 27: Diagrama de flujo de la resolución de dependencias.

constante revisión; y el uso de algoritmos de aprendizaje máquina aumentaría demasiado la complejidad del asunto.

- No proporciona soluciones cuando un árbol no converge.
- A la hora de procesar el árbol, no tiene en cuanta el concepto de compatibilidad en el versionado semántico, pues si finalmente el árbol se queda con una versión, no permite cambiarla por una más alta compatible.
- Obliga a modificar el formato de versiones, permitiendo dejar abiertos valores en las dependencias con las X, lo cual puede ser perjudicial en el resto del programa, al no tener versiones totalmente definidas.

Dado que este sistema tiene una gran serie de inconvenientes y aumenta notablemente la complejidad del programa, se ha decidido dejar su implementación, aunque podría ser tenido en cuenta si en un futuro la relación de dependencias por la evolución propia de los paquetes dentro del departamento se hiciese mucho más compleja y se necesitase algún procesamiento mucho más elaborado.

Por ahora, para conseguir un buen funcionamiento se utilizará una metodología que partirá de la idea del procesado inicial, pero incluyendo una lógica más elaborada. La lógica de esta alternativa se muestra en la figura 27.

El primer paso es obtener la lista de paquetes y dependencias que se hará del mismo modo que se explicó en la alternativa inicial. Se tomará cada paquete del fichero de instalación y

se irán anotando las dependencias. Si este paquete tiene dependencias anidadas, se irá llamando a la función de forma recursiva hasta conseguir la lista entera de dependencias.

Uno de los problemas de este método, que no se ha comentado con anterioridad, es el problema de las referencias circulares. Aunque no deberían de ocurrir si el que define las dependencias lo hace correctamente, podría darse el caso de que un paquete A tenga como dependencia un paquete B, que tenga como dependencia un C, que a su vez tenga como dependencia a A. Al bajar por el árbol y volver a llegar al mismo paquete (todo suponiendo en la misma versión), se produciría un bucle infinito. Este hecho también debe ser previsto por el programa, que en cuanto encuentre este caso, deberá de terminar el proceso y notificar un error.

Una vez de las novedades es que en esta primera fase, también es necesario hacer la distinción entre los tipos de paquetes (entre requeridos (`'requested'`) y dependencias (`'dependence'`). Para obtener el tipo, se utilizará `'requested'` en caso de que el paquete se haya anotado estando en el nivel inicial del árbol, y si se ha obtenido en una rama inferior, se llamará `'dependence'`. Para ilustrar mejor los resultados de los distintos pasos, se utilizará el ejemplo utilizado para la alternativa inicial. Tras el primer paso, la salida obtenida será del tipo:

```
[{'name': 'Paquete 6', 'version': '2.0.0', 'type': 'dependence'},
 {'name': 'Paquete 4', 'version': '1.0.0', 'type': 'dependence'},
 {'name': 'Paquete 5', 'version': '1.1.0', 'type': 'dependence'},
 {'name': 'Paquete 1', 'version': '1.0.0', 'type': 'requested'},
 {'name': 'Paquete 7', 'version': '1.0.0', 'type': 'dependence'},
 {'name': 'Paquete 5', 'version': '1.0.0', 'type': 'dependence'},
 {'name': 'Paquete 2', 'version': '2.0.0', 'type': 'requested'},
 {'name': 'Paquete 7', 'version': '1.0.0', 'type': 'dependence'},
 {'name': 'Paquete 5', 'version': '3.0.0', 'type': 'dependence'},
 {'name': 'Paquete 6', 'version': '2.1.0', 'type': 'dependence'},
 {'name': 'Paquete 3', 'version': '1.5.0', 'type': 'requested'}]
```

El segundo paso consiste en agrupar las versiones del mismo paquete, de modo que quede reducida la lista anterior y sean identificables los conflictos, que pueden ser ocasionados cuando aparezca una versión en más de una ocasión. También podría ocurrir que en la primera lista para un mismo paquete apareciese varias veces la misma versión, lo cual no generaría un conflicto. En esta fase, todas las versiones repetidas se eliminarían, de forma que la lista final solo contenga versiones diferentes del mismo paquete.

Uno de los problemas es que si el paquete apareció una vez como paquete requerido y otra como dependencias, cuál es el tipo de paquete final que se toma. Para resolver este problema, el criterio es que si un paquete aparece en alguna ocasión como requerido, su tipo final será `'requested'` porque aunque aparezca como dependencias, al final el usuario eligió instalar ese paquete de forma voluntaria. Si en todas las versiones aparece como dependencia, entonces el tipo será `'dependence'`. Siguiendo este razonamiento, se puede obtener la siguiente lista de paquetes:

```
[{'name': 'Paquete 6', 'version': ['2.0.0', '2.1.0'], 'type': 'dependence'},
 {'name': 'Paquete 4', 'version': ['1.0.0'], 'type': 'dependence'},
 {'name': 'Paquete 5', 'version': ['1.1.0', '1.0.0', '3.0.0'], 'type': '
    dependence'},
```

```
{'name': 'Paquete 1', 'version': ['1.0.0'], 'type': 'requested'},
{'name': 'Paquete 7', 'version': ['1.0.0'], 'type': 'dependence'},
{'name': 'Paquete 2', 'version': ['2.0.0'], 'type': 'requested'},
{'name': 'Paquete 3', 'version': ['1.5.0'], 'type': 'requested'}]
```

Para continuar el proceso de resolución de dependencias, será necesario hacer uso de la colección de paquetes, que se obtiene como se comentó en el capítulo anterior. Con los datos de esta lista, se empezaría a iterar paquete a paquete resolviendo sus propios conflictos.

La primera de las comprobaciones será si en la lista de paquetes aparece una única versión (lo cual es el escenario ideal puesto que no hay conflicto con ninguna otra). Este caso ocurriría en el ejemplo en los paquetes 4, 1, 7, 2 y 3. En este caso, inicialmente se tomaba directamente la versión que aparecía en la lista. Sin embargo, este hecho no es del todo eficiente porque puede existir una versión totalmente compatible más actual. Por ello, se ha optado por buscar la versión más alta compatible (aquella cuyo primer número dentro de la versión coincide con la versión de la lista y los siguientes son mayores). Si la versión más alta compatible es la misma que la versión que apareció en la lista inicial, no se realizará ninguna operación más y se tomará la versión como definitiva.

En caso de que exista una versión más alta compatible, *myum* tomará por defecto esta versión compatible. El inconveniente que podría tener esta acción es que el usuario podría desear explícitamente la versión que se obtuvo de la resolución de dependencias, y tampoco se desea llegar a una automatización tan extrema como en la segunda alternativa en la que el usuario no tenga oportunidad de opinión. Para ello, se diseñó la opción `-na` (no automático), que en caso de ser activada varía el flujo de la figura 26 y permite un control mayor al usuario. En esta situación aparecería una lista de opciones en las que el usuario podría elegir la opción a tomar. Estas opciones, para este caso serían:

1. Instalar la versión que se obtuvo mediante la resolución de dependencias.
2. Instalar la versión más alta compatible (opción recomendada).
3. Abortar la instalación.
4. Omitir el paquete.

Como se puede apreciar, cuando se ofrezcan opciones al usuario, siempre se añaden como alternativas abortar la instalación, que detendría todo el proceso u omitir el paquete, que borraría el paquete que se está resolviendo de la lista y continuaría con el siguiente. En este escenario, como no hay conflictos, las alternativas son tan sencillas como la versión obtenida previamente o la más alta compatible. Posteriormente se verán casos más complejos donde la variedad de opciones será mayor.

La segunda de las situaciones es que exista más de una versión en la lista. En este caso, en primer lugar, se encuentra el escenario sencillo en el cual todas las versiones son compatibles (como ocurre con el paquete 6, en el que aparecen las versiones 2.0.0 y 2.1.0). El comportamiento será muy similar al caso anterior. El programa buscará en primera instancia la versión más alta compatible (que podría ser la 2.2.0 en el ejemplo) y anotaría ésta como versión definitiva.

De nuevo, para dar más control al usuario se puede optar por elegir la opción `-na`, que en este caso funcionaría de igual modo, pero mostraría más opciones (en este caso, si se supone la versión más alta la 2.2.0) aparecería la siguiente información:

```
Paquete 6 is required in version 2.0.0.
Paquete 6 is required in version 2.1.0.
Choose one of the following options:
1) Install package Paquete 6 2.0.0. (dependence)
2) Install package Paquete 6 2.1.0. (dependence)
3) Highest compatible version available Paquete 6 2.2.0. (recommended)
4) Abort the program.
5) Skip the package.
```

Junto con la información de las versiones, aparece también el tipo de paquete correspondiente a cada versión porque el usuario podría querer dar más importancia a la versión requerida que a una obtenida por dependencias y así puede saber cuál es cada una. La versión más alta compatible aparece siempre como recomendada, excepto cuando sea una de las versiones de la lista, en cuyo caso aparecerá simplemente su tipo.

El último de los casos posibles es que en la lista aparezcan versiones que no sean compatibles (en el ejemplo, lo que ocurre con el paquete 5, que tiene como versiones la 1.1.0, 1.0.0 y la 3.0.0, que aunque las dos primeras lo sean, la tercera es incompatible al resto). En este caso, la opción normal sería preguntar al usuario que resuelva el conflicto presentando una lista de opciones, como en los casos anteriores. El problema que tiene es que se perdería bastante automatización en el programa si cada vez que se quiera instalar lo mismo se necesita indicar cómo resolver el conflicto. Para resolver este problema, aparece un nuevo fichero de metadatos, que no se presentó previamente al no estar en el contexto global del programa, pero que es muy útil para esta situación: el fichero de conflictos.

5.4.1. Fichero de conflictos

Para automatizar la resolución de conflictos entre paquetes no compatibles, se define el fichero `myum_conflict.json`. Este fichero se encargará de almacenar la información acerca de todos los posibles conflictos ocurridos entre versiones incompatibles de un mismo paquete que hayan ocurrido y la última respuesta que dio el usuario para resolverlos. De este modo, si el mismo conflicto vuelve a suceder, se podrá leer la información de este fichero y repetir la elección previa. En caso de que el desarrollador no quedase satisfecho de su elección anterior, al igual que en los demás casos, siempre existe la opción `-na` para salir del flujo habitual y obligar a preguntar de nuevo al usuario. La respuesta que se dé cuando pregunte, en este escenario, debe ser guardada también en el fichero para lecturas posteriores. El formato del fichero de conflictos se muestra en la figura 28.

El conjunto de posibilidades deben coincidir exactamente para que se tomen en cuenta. En la plantilla de ejemplo, aparece que hubo conflictos en el paquete *pkg1* y que cuando las opciones disponibles eran *ver1*, *ver2* y *ver3*, el usuario escogió ver3; y cuando las opciones eran únicamente *ver1* y *ver2*, se prefirió *ver1*.

En el flujo de la resolución de dependencias, cuando se encuentren paquetes no compa-

```
{
    "json_database_version": "2.0.0",
    "conflict": [
                {
                "name" : "pkg1",
                "solutions": [
                        {
                        "possibilities": [ver1, ver2, ver3],
                        "election" : ver3
                        },
                        {
                        "possibilities": [ver1, ver2],
                        "election" : ver1
                        }
                ]
                }
        ]
}
```

Figura 28: Formato del fichero `myum_conflict.json`.

```
myum WARNING: Major version conflict with package basic.
Package basic is required in version 2.0.0
Package basic is required in version 1.0.0
Choose one of the following options:
1) Install package basic 2.0.0. (requested)
2) Install package basic 1.0.0. (dependence)
3) Highest compatible version available basic 1.0.1. (recommended)
4) Abort the program.
5) Skip the package.
```

Figura 29: Resolución de conflictos de versiones incompatibles por el usuario.

tibles, en primer lugar se mirará en este fichero y si para el paquete que da conflictos, existe la misma combinación de versiones que se quieren resolver, se tomará la solución almacenada. En caso contrario, dado que las versiones son incompatibles, no queda más remedio que preguntar al usuario.

En el caso de que se interactúe, la forma de presentar las opciones será similar a lo anterior. En primer lugar se presentarán todas las versiones de la lista original, indicando si son dependencias o son requeridas. A continuación, aparecerán las versiones más altas compatibles (si no han aparecido ya). La principal diferencia es que como ahora hay varias versiones incompatibles, cada una puede tener una versión más alta; por lo que en esta lista se mostrarán varias versiones recomendadas. Por último, aparecen las opciones habituales de abortar y omitir.

Para comprender esto mejor se presentará un ejemplo real. Se supone la existencia del paquete *basic* en las versiones 1.0.0, 1.0.1 y 2.0.0 y que en el fichero de instalación se solicitó su instalación en versión 2.0.0, mientras que otro paquete requerido tiene como dependencia *basic* en su versión 1.0.0 y que no hay ninguna opción guardada y se debe preguntar al usuario. En dicho caso, las opciones disponibles se muestran en la figura 29.

En el escenario mostrado, como las versiones 1.0.0 y 2.0.0 son incompatibles, aparecen

ambas en la lista (la 2.0.0 como solicitada y la 1.0.0 como dependencia). Entre las versiones 1, como existe una versión más alta compatible, la 1.0.1, se muestra como versión recomendada, pero como en las 2 ya está mostrada no se añade información adicional. Las opciones de abortar y omitir aparecen como en todos los casos.

Una vez que el usuario elige una opción, se tomará como solución definitiva para resolver el paquete y se incluirá en el `myum_conflict.json` para posteriores conflictos. Si el usuario decidió abortar u omitir, esta opción también se almacenará y la próxima vez se tomará dicha acción. En caso de que el usuario no quiera usar el fichero, con la opción `-na`, como se ha comentado, se puede pedir que se pregunte y en caso de que el conflicto existiera en el fichero, la nueva selección sobrescribiría la anterior por si en una instalación posterior se necesitase utilizar.

Cuando termine este proceso para cada paquete, se deberá obtener una lista final con todos los paquetes resueltos en donde solo aparezca una versión posible. Tras ejecutar el proceso anterior, es posible que algunos paquetes tengan como versión una `'S'`, que marca que el paquete se ha omitido. En esos casos, se deberá eliminar dichos paquetes para que no sean tenidos en cuenta. Una vez hecho esto, se deberá obtener una lista final con el siguiente formato (se ha supuesto que el paquete 5 se omite y el resto se resuelven de forma automática y no existen versiones más altas compatibles):

```
[{'name': 'Paquete 6', 'version': '2.1.0', 'type': 'dependence'},
 {'name': 'Paquete 4', 'version': '1.0.0', 'type': 'dependence'},
 {'name': 'Paquete 1', 'version': '1.0.0', 'type': 'requested'},
 {'name': 'Paquete 7', 'version': '1.0.0', 'type': 'dependence'},
 {'name': 'Paquete 2', 'version': '2.0.0', 'type': 'requested'},
 {'name': 'Paquete 3', 'version': '1.5.0', 'type': 'requested'}]
```

5.5. Preparación de la lista de instalación final I: Análisis de los paquetes instalados.

Cuando se hayan resuelto las dependencias se dispondrá de una lista de paquetes para instalar. Sin embargo, esa lista no puede ser proporcionada todavía a las funciones de instalación porque puede ocurrir que existan paquetes ya instalados, por ejemplo. En esos casos es necesario definir un comportamiento que decida si se debe volver a instalar el paquete o si se debe mantener como está. Para poder realizar esta tarea, se utilizará por primera vez el fichero de datos de instalación (`.myum.db`) si existe. En el caso de que se haya utilizado el programa por primera vez y no se haya generado siquiera el fichero, se considerará que no hay nada instalado y entonces, la lista obtenida al resolver las dependencias se convertirá directamente en la lista final de instalación.

Con el fichero de datos de datos de instalación se buscará entre la lista de paquetes instalados cada paquete de la lista obtenida al resolver las dependencias y se pasará la primera comprobación únicamente en caso de que el paquete no esté instalado o la versión a la que se vaya a instalar sea más alta y compatible que la versión actual instalada. De este modo se excluyen, en principio, los siguientes casos:

1. Se pide instalar un paquete en la misma versión en la que ya está instalado (reinstalación).
2. Se pide instalar un paquete en una versión más alta incompatible con la instalada.
3. Se pide instalar un paquete en una versión más antigua (compatible o no) con la instalada (*downgrade*).

En el caso de que se produzca una de las tres circunstancias anteriores, el paquete será, en general, eliminado de la lista final de instalación. Sin embargo, no sería bueno imposibilitar de forma total la reinstalación de un paquete puesto que podría ser que el usuario lo modificara e hiciera algún cambio no deseado y quisiera volver a la versión que tenía. Del mismo modo, tampoco es buena idea bloquear la actualización a una versión incompatible porque lo lógico es que al final, todo el *software* evolucione y en el caso de bajar de versión también puede ser interesante en algún caso en el que una versión más alta no ofrezca el rendimiento deseado.

Obviamente, una manera sencilla de resolver el problema es desinstalar un paquete para que no aparezca como instalado y volver a ejecutar la instalación. El problema es que requiere más pasos que se pueden agrupar. Para resolver esto, *myum* vuelve a hacer uso del *flag* `-f` (forzar). Puesto que salvo que la versión sea más alta y compatible, una instalación puede no tener el efecto deseado, en principio se adopta el comportamiento descrito, pero en caso de que el usuario realmente quiera saltarse cualquiera de las tres normas y realizar la instalación, puede realizar el forzado. El problema es que forzar la instalación puede ser peligroso puesto que se omitirían todas las comprobaciones para instalar finalmente el paquete, por lo que se debe estar bastante seguro cuando se utiliza. En caso contrario, siempre desinstalar y volver a instalar es mucho más seguro. Para el caso de forzar la instalación, en realidad las comprobaciones no se omitirían, si no sus resultados, por lo que en caso de error, los mensajes de advertencia se seguirían mostrando, aunque no tendrían efecto.

5.6. Preparación de la lista de instalación final II: Comprobación de la integridad

Cuando un paquete instalado supere las comprobaciones expuestas en la sección anterior, antes de poder ser definitivamente considerado para su instalación, debe superar el *test* de integridad. Esta comprobación trata de ver si el contenido que se instaló previamente ha sido modificado o no; y si los ficheros originales siguen existiendo y no se haya añadido ninguno nuevo. Para realizar estas comprobaciones, se utiliza la función *hash MD5*. En el próximo capítulo se pormenorizará el aspecto de la comprobación de paquetes cuando se explique la opción que se encarga de realizar específicamente esta comprobación para un paquete instalado (`-ck`).

Por ahora, para entender la instalación, lo importante es entender que comprueba la integridad del paquete instalado y que en caso de que el resultado del *test* sea negativo, no continuará la instalación a menos de que se haya forzado la instalación, en cuyo caso, se mostrarán las advertencias de los ficheros que se han modificado, borrado o añadido y se proseguirá con el proceso de instalación.

Una vez que un paquete supere todas las comprobaciones hasta aquí será instalado. El

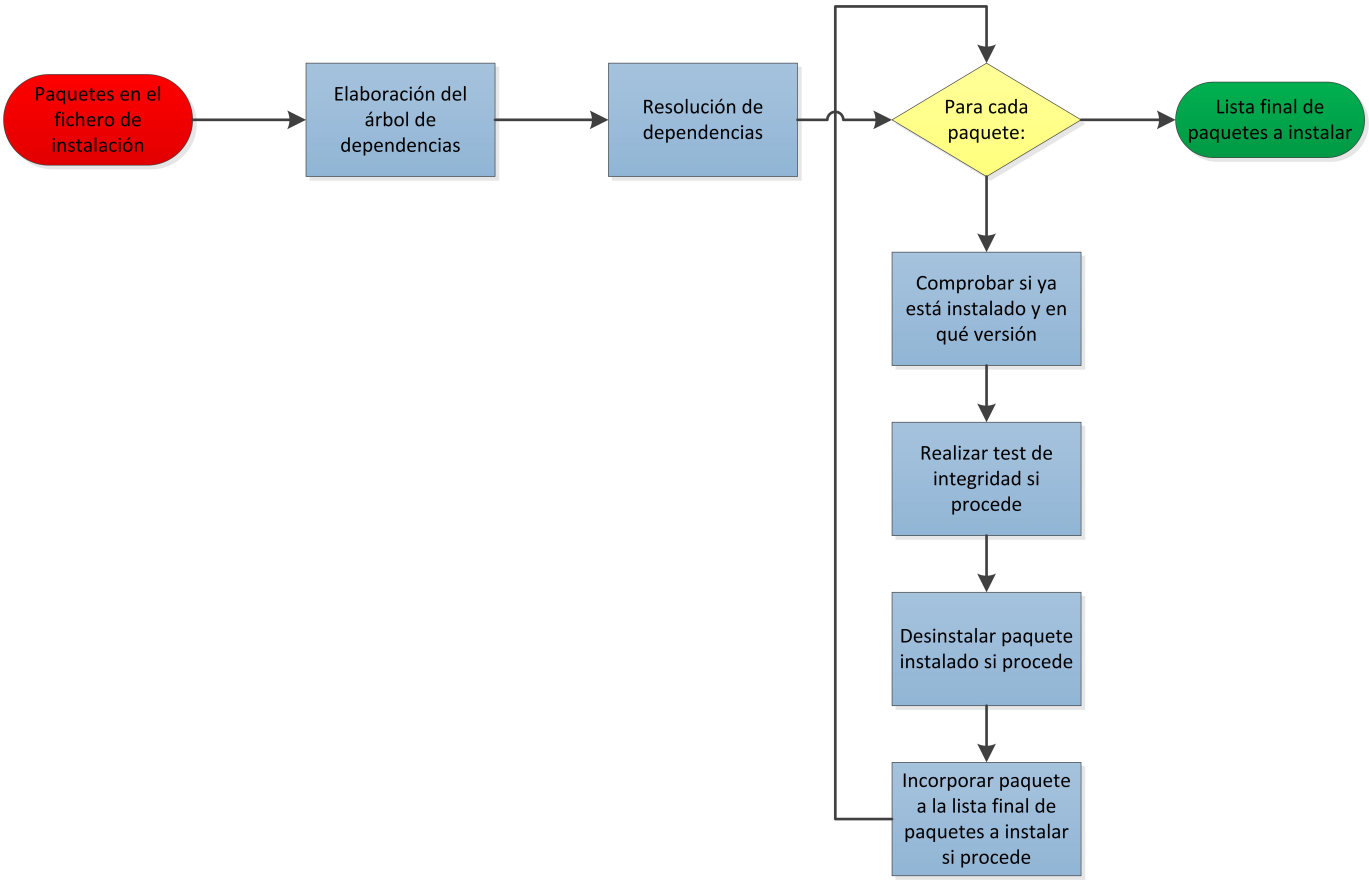

Figura 30: Proceso de obtención de la lista de paquetes a instalar.

programa, como paso adicional, dispone del mecanismo a través de una variable para distinguir entre si un paquete va a ser instalado o actualizado o reinstalado para registrar la operación adecuada en el histórico de operaciones. El formato de la lista de instalación será idéntico al de la lista obtenida tras resolver las dependencias, con la única salvedad de que aparecerá una clave llamada `'mode'`, para hacer la distinción comentada. El valor de esta clave será 0 si el paquete es instalado o 2 si es reinstalado/actualizado. El valor 1 se reserva para el caso en el que el paquete no se instala y por tanto desaparece de la lista final de paquetes. Se ha decidido utilizar un número superior para la actualización/reinstalación porque en muchas ocasiones proviene de usar el forzado (la reinstalación solo se puede llevar a cabo de ese modo). La lista anterior de paquetes tras resolver las dependencias puede quedar en la siguiente lista y el proceso completo para la obtención de la lista final de paquetes se puede resumir en la figura 30.

```
[{'name': 'Paquete 6', 'version': '2.1.0', 'type': 'dependence', 'mode': 2},
 {'name': 'Paquete 4', 'version': '1.0.0', 'type': 'dependence', 'mode': 0},
 {'name': 'Paquete 1', 'version': '1.0.0', 'type': 'requested', 'mode': 0},
 {'name': 'Paquete 3', 'version': '1.5.0', 'type': 'requested', 'mode': 2}]
```

5.7. Desinstalación de paquetes

Antes de proceder a la instalación propiamente dicha, como se puede ver en el esquema de la figura 30, hay un paso más adicional: la desinstalación de paquetes. En el caso de que un paquete esté instalado, pero finalmente se vaya a instalar de nuevo es necesario borrar los datos instalados para que en su lugar se incorporen los nuevos datos.

Esta desinstalación tiene un rasgo bastante particular y es que a diferencia del resto, en principio no se registra en el historial de operaciones, ni siquiera elimina al paquete de la lista de paquetes instalados. La razón es que si lo hiciera, si un paquete se actualizase aparecería un registro de 'desinstalar' seguido de otro de 'actualizar', lo cual no tiene mucho sentido, pues si un paquete es desinstalado no puede ser actualizado después. La eliminación de la lista de paquetes instalados se podría hacer, pero requeriría un procesamiento adicional innecesario, puesto que tanto el histórico como la lista de ficheros instalados se encuentran en el mismo fichero, y si se hace posteriormente cuando termine la instalación el registro de la operación y en este caso, la modificación de la instalación, se puede ahorrar tener que escribir el fichero dos veces (con una vez al final bastaría).

La práctica del programa, sin embargo, muestra un claro inconveniente de no realizar estos registros de desinstalación. El problema ocurre en el caso de que la desinstalación se efectúe correctamente, pero después ocurra un problema en la instalación que termine el proceso. En dicho caso, el paquete seguiría apareciendo como instalado porque no se modificó el fichero de instalación pero realmente estaría desinstalado porque el proceso quedó interrumpido. Para solucionar esto, se ha ideado un sistema de *backup* durante la instalación.

5.7.1. Sistema de *backup* durante la instalación

Uno de los principales problemas de la actualización o reinstalación de paquetes es el hecho de tener que eliminar la versión actual para poder ser reemplazada. El mayor problema que supone es la presencia de un error durante la instalación, que normalmente puede ocurrir porque alguno de los paquetes que se vayan a instalar requieran unas fuentes que no estén disponibles en el repositorio, bien porque no existan o porque en el momento de la instalación se produzca un fallo en el servidor y no se pueda establecer la conexión.

Para protegerse de este problema es necesario almacenar de algún modo los paquetes que se desinstalaron con el fin de una actualización o reinstalación posterior. Para hacer esto, se podría utilizar una estructura que realizase dicho almacén, aunque dado que se dispone de un fichero que puede registrar las operaciones (`.myum.db`), lo más sensato es hacer uso de él. El único problema es que no se puede modificar directamente sobre él porque si no, en caso de éxito, aparecerían registros no deseados.

Para poder hacer uso de la estructura sin modificar el fichero, la táctica es hacer una copia del `.myum.db`, a la que se llamará `.myum.bak`, en la que se podrán anotar todos los registros necesarios, de modo que cada vez que se desinstale un paquete para poderle reinstalar o actualizar, en la copia de *backup* se anotará el registro de desinstalación tal y como si de una operación de borrado normal se tratase y se eliminará el paquete de la lista de paquetes

instalados.

Cuando llegue el momento de la actualización o reinstalación, la operación quedará registrada tanto en el fichero original como en el de *backup* y en caso de éxito, como el fichero original tiene registradas todas las operaciones finales, el otro fichero ya no será necesario y será eliminado.

Si ocurre un error en mitad de la instalación, mientras que el fichero original no habrá registrado todo lo que haya ocurrido, el de *backup* sí, por lo que tras parar el proceso de instalación, se elimina el fichero original y se establece como `.myum.db` el fichero de *backup*, que al tener todos los registros se encuentra en un estado consistente. La única desventaja que tendría es que para los paquetes que se instalaron correctamente aparecerá un registro de borrado y otro de instalación en el historial, pero la lista final de paquetes instalados será correcta y para los paquetes que no llegaron a instalarse se reflejará el borrado y se sabrá que ya los ficheros no se encuentran en el sistema.

Tras llegar a este punto, lo que se podría plantear es si sería conveniente volver al estado inicial antes de comenzar la instalación o si dejar los paquetes tal y como están tras producirse el error. El hecho de volver al inicio es interesante por el hecho de poder volver a un punto de partida conocido. El principal inconveniente es que si el usuario ha pedido realizar ciertas operaciones es porque tiene intención de llevarlas a cabo y si ha solicitado la instalación de 10 paquetes y consigue instalar 9 correctamente, no hay ningún motivo para tener que eliminar lo que se requirió en la instalación y terminó con éxito. Es más, incluso hasta el paquete que se desinstaló y al actualizarse falló, si bien puede ser que la solución final sea volver al estado anterior, una posibilidad es que se decida intentar ver el error y volver a intentarlo a instalar (sin la necesidad de volver a realizar la instalación). Si es el caso de que el servidor está fallando durante un tiempo, se puede esperar a que se reestablezca y volver a intentarlo. Y aun en el peor de los casos de que no se pudiera instalar, si un usuario tiene la versión 1.1.0 y pide la 1.3.0 y no puede, es posible que al final pida la 1.2.0, en vez de la que tenía. Por ello, la decisión es que si ocurre un error, que se recupere el programa a un estado consistente, pero no se eliminen los cambios realizados.

5.8. Copia de ficheros

Los pasos descritos anteriormente durante el proceso de instalación únicamente servían para preparar la misma, pero no es hasta este momento cuando se puede hacer efectiva. Si se han realizado correctamente todos los pasos anteriores se tendrá una lista definitiva para proporcionar al instalador, en la que como se mostró aparecerá el nombre del paquete, su versión; y el tipo (solicitado o por dependencia) y el modo (instalación o actualización/reinstalación) para ser posteriormente escrito en el registro de instalación.

Para realizar la instalación de cada paquete a partir de este punto, lo primero será obtener el fichero `paquete.json`, que diga el contenido de los datos. Este fichero, aunque se podría encontrar localmente en el directorio `"tools/myum"` dentro del directorio de trabajo, lo normal es que se encuentre en dicha ruta dentro del repositorio donde se encuentre el paquete (obtenido inicialmente al obtener la colección de paquetes). En dicho caso, el programa se

encarga de obtener el fichero haciendo una llamada a *Git* y de ese fichero ahora la parte que se usará será la que aparece dentro de la clave `"data"` (previamente, se utilizó la clave `"dependency"` para extraer las dependencias).

Dentro de esta clave, aparece una lista del contenido a copiar. Cada elemento de la lista es un diccionario con los siguientes campos:

- `'dest'`: Indica el lugar en el que se deben dejar los ficheros instalados partiendo del directorio de trabajo.
- `'source'`: Indica dónde se encuentra el contenido que se quiere instalar en el repositorio donde se encuentra el paquete (en el *tag* correspondiente al paquete).
- `'type'`: Indica el tipo de contenido que se va a copiar. Puede tener dos posibles valores: `"dir"` si se va a copiar un directorio entero o `"file"` si es un fichero independiente.
- `'chmod'`: Indica el tipo de permisos (lectura, escritura y ejecución) que se deben asignar al contenido cuando se descargue en el destino indicado.

Para poder copiar el contenido, solo serán necesarios los tres primeros campos (el cuarto se utilizará posteriormente). La descarga se hará utilizando el comando `git archive` [1] con el siguiente formato:

Para directorios:

```
git archive --remote:URL_repositorio --output:lugar_donde_se_deja_tarball --
    format=tar mtag_nombre_a.b.c:valor_del_source
```

Para ficheros:

```
git archive --remote:URL_repositorio --output:lugar_donde_se_deja_tarball --
    format=tar mtag_nombre_a.b.c valor_del_source
```

La única diferencia para ficheros y directorios son los dos puntos (`:`) que aparecen entre el `tag` y la fuente.

Mediante el comando comentado no se realizará la copia de los ficheros directamente al lugar indicado, sino que se creará un fichero comprimido en formato `.tar` (conocido habitualmente como *tarball*) que contendrá todo el contenido contenido en el `"source"` indicado. Este fichero se dejará en un directorio caché que utilizará el programa para ir dejando todos los datos intermedios que utiliza el programa. La ruta del directorio caché desde el directorio de trabajo es `work/myum`.

Por último, el *tarball* se descomprime en directorio indicado en el campo `"dest"` partiendo del directorio de trabajo. La descompresión se lleva a cabo utilizando la librería de Python `"tarfile"` [14], por lo que no es necesario disponer de un *software* específico para la compresión/descompresión, al contrario que la versión inicial del programa que hacía una llamada al comando `"tar"` que podía o no estar disponible.

Propietario	Grupo	Resto
rwx	rwx	rwx

Tabla 3: Atributos de permisos en Linux.

5.9. Establecimiento de permisos

Uno de los problemas que podrían surgir una vez copiados los ficheros a su lugar correspondiente es que el usuario no pudiera ejecutarlos o que no pudiera editarlos, lo que provocaría que no se pudiera adaptar un paquete para un proyecto determinado. El problema vendría dado porque los permisos que se hayan establecido en los ficheros descargados no sean los adecuados. Por ello, para garantizar el buen funcionamiento, el programa se encargará de establecer los permisos.

Para esta tarea se utilizará el valor de la cuarta de las claves que se presentaban para cada elemento de la lista correspondientes a la clave `'data'`, el `'chmod'`. El valor de este campo es un número octal de tres dígitos que definen los permisos según el sistema de permisos de Linux [18].

Según este sistema de permisos, se definen 9 caracteres de atributos, que representan los permisos de lectura, escritura y ejecución para el propietario, para el grupo y para el resto. El primer bloque es para el propietario, el segundo para el grupo y el tercero para el resto. Dentro de cada bloque, cada uno de los tres dígitos, `'r'`, `'w'` y `'x'` sirven para asignar permisos diferentes de acuerdo con la tabla 3.

El criterio utilizado para establecer los permisos es utilizar un número octal de tres dígitos, de modo que cada '0' o '1' de cada dígito octal represente si se activa un atributo o no. De este modo, un 5, que en binario es '101', significaría que se activan los permisos de lectura y ejecución, pero no los de escritura. Así, el campo `'chmod'` podría tener un valor de 755 en caso de que el propietario tenga todos los permisos y el grupo y resto tengan únicamente de lectura y escritura.

Myum tomará el valor del `'chmod'` y lo aplicará a todos los directorios y ficheros que encuentre en la ruta dada en caso de que `'type'` sea el de directorios, o si el tipo es el de fichero, establecerá únicamente los permisos del fichero.

Las limitaciones que tiene esta funcionalidad es que los permisos en *Windows* no funcionan del mismo modo y no se admiten las separaciones entre propietario, grupo y resto que se presentaban. Por ello, si se da el caso de una instalación en *Windows*, se leerán los permisos y mediante las librerías de *Python*, se establecerán los permisos de la forma que sean más compatibles entre ellos (que será estableciendo únicamente los permisos de lectura y escritura, que son los que permite *Windows*).

Atributo	Ficheros	Directorios
`r`	Permite abrir un fichero y leerlo.	Permite que el contenido de un directorio pueda ser listado siempre que el atributo de ejecución esté activo.
`w`	Permite que un fichero pueda ser escrito o truncado, aunque no permite que los archivos sean borrados o renombrados, cuyo poder está determinado por los atributos del directorio del fichero.	Permite que los archivos dentro de un directorio puedan ser creados, eliminados y renombrados siempre que el atributo de ejecución esté activo.
`x`	Permite que un fichero pueda ser tratado como un programa y pueda ser ejecutado. Los ficheros de programa escritos en lenguajes de *scripting* deben tener también el atributo de lectura para ser ejecutados.	Permite la entrada a un directorio (por ejemplo, mediante `cd directory`).

Tabla 4: Descripción de los atributos de permisos en Linux. Fuente: The Linux Command Line [18].

5.10. Ejecución de los comandos de post-instalación

Tras establecer los permisos en los distintos ficheros la instalación está ya casi completada, pues todos los ficheros están en su correspondiente lugar. No obstante, todavía quedan algunos pasos para completar el proceso. Uno de esos pasos es el de la ejecución de comandos.

Cuando se instala un paquete, como este contiene una cantidad importante de código (mucho de él en *VHDL*, que es un lenguaje bastante conocido de descripción de hardware), es necesaria su compilación para poder realizar la ejecución. En el fichero de definición del paquete existe una clave llamada `'cmd'`, que contiene una lista de comandos que ejecutarán al final de la instalación (tomando como referencia al directorio de trabajo, si estos incluyen direcciones relativas).

Actualmente, los paquetes ejecutan un fichero de instalación llamado `paquete_post_install.py`, que se encarga de definir las variables de entorno necesarias para el funcionamiento del paquete, así como de la compilación del código. Este mismo código tiene en cuenta las distintas plataformas en las que pueda ser instalado el programa para poder tener en cuenta que unos mismos comandos pueden ser ejecutados de una forma u otra dependiendo del sistema operativo. Durante la ejecución de los comandos, *myum* reporta un mensaje indicando si los comandos se ejecutan correctamente o si se ha producido un error durante su ejecución.

5.11. Actualización del fichero de datos de la instalación

Después de ejecutar los comandos adecuados para instalar el paquete, éste está en plenas condiciones para ser utilizado. No obstante, la herramienta debe tener constancia de la instalación del paquete para que en un futuro se puedan realizar más operaciones con él.

Este paso se compone de dos partes: de la inclusión del paquete en la lista de paquetes instalados y del registro de la operación.

Para la inclusión en la lista de paquete instalados, se creará una nueva entrada en la que aparecerá el nombre del paquete, la versión, el tipo (solicitado o por dependencia), la fecha y hora de instalación, y la dirección en la que se encuentran todos los ficheros que se han instalado con el paquete (que será útil para la comprobación de la integridad).

Sobre el registro de operaciones, se incluyen los mismos que para el registro anterior, a los cuales se añade el tipo de operación, que podrá ser `'install'`, si el paquete anteriormente no se encontraba instalado y se instala, `'reinstall'`, si el paquete se vuelve a instalar en la misma versión, o `'update'`, si el paquete se instala en una versión diferente a la que estaba previamente instalada.

En este proceso se tomará el contenido anterior y junto con este, se volverá a editar el fichero `.myum.db` para que mantenga todos los datos actualizados. Además, como se comentó previamente, al fichero de *backup*, `.myum.bak`, también se le realizarán las mismas operaciones para que este refleje los últimos cambios y pueda ser utilizado en caso de error. En caso de ser la primera instalación y que no exista ningún fichero previo, el programa se encarga de generarlo e incluir el contenido.

5.12. Actualización del fichero de instalación

El último paso, que no siempre es necesario llevar a cabo es la actualización del fichero de instalación. Si el usuario utiliza dicho fichero para solicitar la instalación, al final de la misma tendrá instalado aquello que solicitó, por lo que en principio no es necesario actualizar nada. Sin embargo, si se utilizan otros métodos para instalar, que se analizarán posteriormente, que no utilizan el fichero de instalación, sí es interesante que al final los paquetes que se han instalado aparezcan en dicho fichero representando la configuración de la instalación.

Por ello, antes de terminar, se revisan todos los paquetes que aparecen en la lista de paquetes instalados, y aquellos que aparezcan con el tipo `'requested'` que no aparezcan en el fichero de instalación se añaden, de modo que ese fichero si el usuario abre el mismo proyecto desde el inicio en otro lugar, pueda usar el mismo fichero de instalación y obtener la misma configuración o una muy similar (la resolución de dependencias podría ser distinta si aparece una versión nueva compatible o si no se dispusiese del fichero de resolución de conflictos y se escogiese una nueva opción, por ejemplo).

Para una instalación habitual, en principio el fichero no cambia porque se utiliza el fichero de instalación para instalar, salvo que existiesen paquetes ya instalados, pero que sin

haberlos desinstalado, se hayan eliminado del fichero de instalación, en cuyo caso reaparecerían para hacer constancia de que la configuración de instalación actual los tiene presentes. También, podría haber algún cambio en el caso de que un paquete se solicitase en una versión y tras resolver dependencias finalmente se instalase en otra, en cuyo caso aparecería el mismo paquete en el fichero, pero en la versión final instalada.

Para terminar, si todo ha ido con éxito también se elimina el fichero de *backup*, `.myum.bak`, puesto que ya no es necesario al estar registradas todas las operaciones en el fichero de datos de instalación `.myum.db`.

5.13. Ejemplo de instalación

Para concluir este capítulo se presentará un ejemplo de instalación. La figura 31 muestra esta instalación. En esta instalación, se solicita instalar el paquete *basic* en versión 1.0.2 (el paquete ya está instalado) y las dependencias del paquete *apb*, que en este ejemplo son *registers* 1.0.0 , *obt_setup* 1.0.0 y *basic* 1.0.1. Se supone que hay cambios en el repositorio, por lo que se utiliza el forzado para omitir el resultado de la comprobación. Posteriormente se realiza la resolución de dependencias, en la que el único conflicto está entre el paquete *basic*, puesto que se solicitó en versión 1.0.2, pero es dependencia del *apb* en versión 1.0.1. Como por defecto se utiliza el modo automático y las dos versiones son compatibles, se toma la más alta (1.0.2). Posteriormente se comprueba la integridad del paquete *basic* porque ya está instalado y se muestra la lista final de paquetes a instalar donde aparecen por primera vez las dependencias del *apb*, sin contar *basic*. Este paquete, al ser dependencia pero haberse también solicitado aparece con el tipo solicitado.

A continuación, se realiza la instalación y se ejecutan los comandos correspondientes. En este ejemplo se han introducido errores de sintaxis en los *scripts* de post-instalación de los paquetes *registers* y *obt_setup* para mostrar que los paquetes se pueden instalar correctamente pero la compilación de los ficheros llevada a cabo en la post-instalación podría fallar. En ese caso los paquetes tendrían todo el contenido instalado, pero el usuario debería ser consciente y revisar lo que falte por ejecutar o ejecutar el comando con la opción de *debug* para ver qué falla y detectar el problema. En muchos casos el error de ejecución puede ser porque se ejecute un comando de un programa que no esté instalado en el equipo. Los comandos del paquete *basic* sí se ejecutan con éxito y finalmente se reescribe el fichero de definición de la instalación con la información de los nuevos paquetes instalados, aunque en este caso como dicho fichero ya contenía *apb* con `d.d.d` y *basic* con 1.0.2 no habría cambios. Si se utilizase la resolución manual y el paquete *basic* se hubiese preferido en versión 1.0.1, en este paso se hubiese modificado la versión solicitada de *basic* para que en una futura instalación ya no existiera la ambigüedad (aunque al elegirse una anterior también se intentaría actualizar a la más alta compatible). La figura 32 muestra el comienzo de la instalación en modo no automático en el cual el usuario elige la versión 1.0.1. En esta figura el proceso de instalación no se muestra completo, pero como una vez que se resuelven las dependencias el proceso no cambia, la parte final es idéntica a la de la figura 31.

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -i -f
myum INFO: Workspace is /home/projects/cr_tools_myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is /home/eda/myum/myum_mirror_list.json
myum WARNING: Working repository has uncommitted changes.
myum INFO: The following packages were required to install:
1    apb                Only dependencies
2    basic              1.0.2     (requested)

myum WARNING: There are different required versions for package basic: ['1.0.1', '1.0.2']. Myum will take the highest compatible if possible.
myum INFO: Package basic version 1.0.2 is already installed.
myum INFO: Checking package basic integrity...
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic.doxygen OK.
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic_doxygen.bat OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin.svg OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.btim OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.svg OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo.vhd OK.
myum INFO: File design/devices/script/hds/crip_basic.src.hdp OK.
myum INFO: File tools/myum/basic_post_install.py OK.
myum INFO: Package basic version 1.0.2 will be reinstalled.
myum INFO: The following packages will be installed after solving dependencies:
1    registers          1.0.0     (dependence)
2    obt_setup          1.0.0     (dependence)
3    basic              1.0.2     (requested)

myum INFO: Installing packages...
myum INFO: Installing package registers version 1.0.0 ...
myum INFO: Executing commands for package registers version 1.0.0 ...
myum WARNING: Command 'python3 tools/myum/registers_post_install.py' was not executed properly.
myum INFO: Package registers version 1.0.0 was successfully installed.
myum INFO: Installing package obt_setup version 1.0.0 ...
myum INFO: Executing commands for package obt_setup version 1.0.0 ...
myum WARNING: Command 'python3 tools/myum/obt_setup_post_install.py' was not executed properly.
myum INFO: Package obt_setup version 1.0.0 was successfully installed.
myum INFO: Installing package basic version 1.0.2 ...
myum INFO: Executing commands for package basic version 1.0.2 ...
myum INFO: Command python3 tools/myum/basic_post_install.py execution was successfully executed.
myum INFO: Package basic version 1.0.2 was successfully installed.
myum INFO: Installation completed!
myum INFO: Rewritting install definition file...
myum INFO: Finished!!
```

Figura 31: Instalación de paquetes.

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -i -f -dr -na
myum INFO: Workspace is /home/projects/cr_tools_myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is /home/eda/myum/myum_mirror_list.json
myum WARNING: Working repository has uncommitted changes.
myum INFO: The following packages were required to install:
1    basic              1.0.2     (requested)
2    apb                Only dependencies

myum WARNING: Major version conflict with package basic.
Package basic is required in version 1.0.2
Package basic is required in version 1.0.1
Choose one of the following options:
1) Install package basic 1.0.2. (requested)
2) Install package basic 1.0.1. (dependence)
3) Abort the program.
4) Skip the package.
2
myum WARNING: Package basic is already installed in version 1.0.2 and version required 1.0.1 is older.
myum INFO: Checking package basic integrity...
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic.doxygen OK.
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic_doxygen.bat OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin.svg OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.btim OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.svg OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo.vhd OK.
```

Figura 32: Instalación de paquetes con resolución no automática de conflictos.

6. Funcionalidades del programa complementarias

En los dos capítulos anteriores se han presentado las funcionalidades básicas para conseguir que los paquetes queden instalados dentro de un proyecto. En realidad, aunque la principal opción sea la de instalar, existen otras también muy necesarias para el conjunto del programa. En este capítulo se analizarán dichas funcionalidades.

6.1. Comprobación de la integridad de un paquete

Durante el proceso de instalación, una de las comprobaciones que se realizan cuando un paquete está instalado y se le solicita una actualización o reinstalación es la comprobación de la integridad. Aparte de poder realizar la comprobación durante el proceso de instalación, *myum* ofrece la posibilidad de realizar el *test* de integridad a un paquete instalado mediante la opción `-ck` (o alternativamente `--check`).

Para realizar las comprobaciones se utilizará la función de *hash MD5* [19], que es uno de los principales métodos para la comprobación de ficheros. Esta función permite comprimir cualquier longitud de datos en un resumen de 128 bits. A nivel criptográfico, el *hash MD5* no es seguro y se pueden encontrar colisiones (distintos mensajes que generen el mismo *hash*) en pocos segundos. De hecho, el *NIST* (*National Institute of Standards and Technology*) [20] recomienda que los algoritmos de *hash* utilizados sean como mínimo de la familia de *SHA-2*. Sin embargo, para la aplicación de verificación de ficheros no es esencial la capacidad criptográfica del algoritmo, sino conseguir que para dos ficheros distintos se obtengan resúmenes distintos y teniendo en cuenta el tamaño del resumen, la probabilidad de colisión es de $\frac{1}{2^{128}}$.

Originalmente, el *myum* calculaba el *hash* de todo el contenido del paquete y lo almacenaba en un fichero `.md5` en el repositorio, de modo que si se quería comprobar la integridad del paquete, se calculase el resumen del paquete instalado y se comparase con el valor del repositorio. Este sistema tiene varios inconvenientes:

- Si no se supera el *test* de integridad, simplemente te puede indicar él resultado negativo, pero no el fichero concreto que ha sufrido modificaciones. En el contexto del programa, esta funcionalidad es muy útil porque permite ver qué ficheros ha modificado el desarrollador y a la hora de tomar una decisión de instalación, poder considerar si los cambios deben ser preservados o no, mientras que del modo inicial si hay centenares de archivos es muy difícil encontrar qué fichero falló.
- Al igual que este método no reporta cambios en ficheros individuales, si todos los ficheros se mantienen pero aparece uno nuevo o se elimina uno, la comprobación también sería errónea y no se podría conocer dichos cambios.
- Obliga a mantener un fichero en el repositorio con el valor del *hash*, lo cual puede conllevar el problema de que alguien cuando desarrolle su proyecto se le olvide o desconozca que debe subir un fichero con el *MD5* para la comprobación posterior.

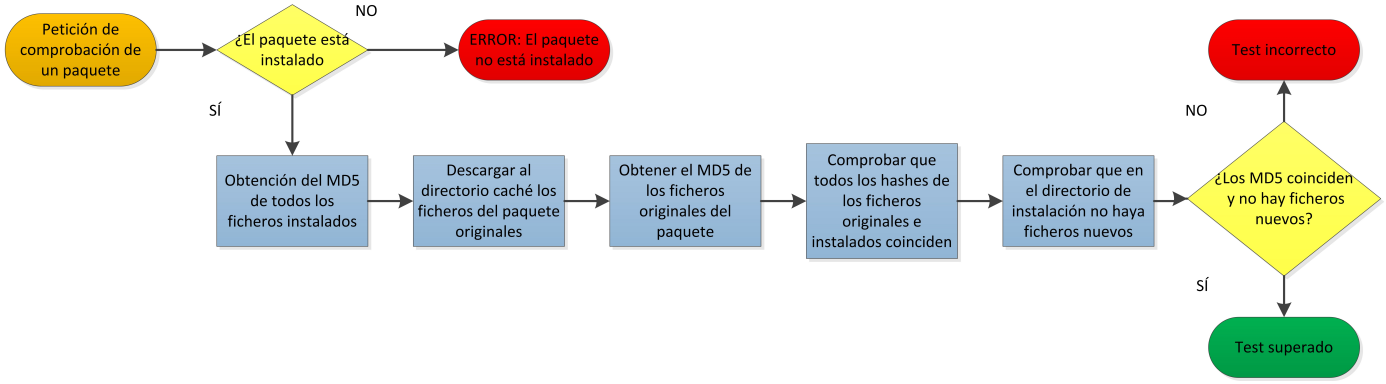

Figura 33: Proceso de comprobación de un paquete.

- Si se actualiza el contenido del repositorio porque se corrige algún fallo, pero no se actualiza el *hash*, el resultado de la comprobación será siempre negativo.

Para prevenirse de estos problemas, la alternativa pretende utilizar un sistema que calcule los *MD5* de los ficheros locales y los del repositorio en el acto, de modo que no se necesite almacenar ningún archivo que deba mantenerse actualizado, y que permita indicar dónde se producen los fallos en la comprobación. La figura 33 muestra el proceso.

En esta alternativa será donde se utilice de forma razonada la información de los ficheros instalados presentes en el fichero de datos de la instalación. En primer lugar, antes de comprobar un paquete se comprueba si está instalado utilizando dicho fichero y en caso negativo se termina el proceso (como se verá más adelante en la versión gráfica, en este modo si un paquete no está instalado ni siquiera será posible pedir esta comprobación). Posteriormente, tomando la lista de ficheros instalados se creará una nueva lista en la que aparecerá para cada fichero su *hash MD5*. El formato que presenta esta lista es el siguiente:

```
[{'hash': '43bad6241c0146d03049790cdb61a337', 'file': 'design/devices/source/
    crip_basic/doxygen/crip_basic.doxygen'},
{'hash': 'be82a82b7ed718f4d8f765d97b42e2a9', 'file': 'design/devices/source/
    crip_basic/doxygen/crip_basic_doxygen.bat'},
{'hash': 'ed4ac37b503cc6826dedd6b95d4a1b69', 'file': 'design/devices/source/
    crip_basic/doxygen/ddrin.svg'},
{'hash': '6f309bb196e65b94b77485bcb7b20047', 'file': 'design/devices/source/
    crip_basic/doxygen/ddrin_wave.btim'}]
```

A continuación, y a diferencia de la alternativa inicial, se descargará el contenido del paquete al directorio caché como si se tratara de una instalación para así poder disponer del contenido actual del repositorio y poder hallar sus resúmenes. Para realizar dicho cálculo, se podrían calcular los *hashes* de cada uno de los ficheros descargados, pero el objetivo es poder hacer una comparación entre los resúmenes locales y remotos y en caso de que en un lugar faltaran o hubiera ficheros nuevos, las dos listas podrían no tener el mismo orden o tener un número distinto de elementos.

Para solucionar este problema, se toma la lista inicial de ficheros obtenida del fichero de datos y se calculan los *hashes* de los ficheros descargados que coinciden con los nombres de esa lista, de manera que se puedan tener dos listas con los mismos elementos ordenados

para poder realizar una comparación directa. En caso de que algún fichero falte se establecerá un -1 para que la lista tenga el mismo número de elementos. Este caso normalmente ocurrirá cuando localmente se borre algún archivo, pues si en el fichero de datos se escribieron los ficheros originales del repositorio, lo lógico es que estos se preserven. Con las dos listas preparadas, se pueden identificar los ficheros que se hayan modificado localmente o se hayan eliminado y se puede reportar de forma individual en qué fichero los resúmenes no coinciden, al contrario que con la alternativa inicial.

Hasta lo presentado, todavía quedaría un caso importante que cubrir: la posibilidad de que se añadan ficheros nuevos. Para cubrir este caso, se lee del fichero de datos del paquete (`paquete.json`) los datos referentes a los directorios y ficheros que se deben instalar y se realiza una búsqueda recursiva en dichos directorios locales de todos los ficheros actuales, la cual se compara con la que aparece en el fichero de datos de la instalación para conocer qué ficheros se han añadido nuevos.

Con esta idea general básica, en principio, en *Linux* no se experimentan problemas, pero en *Windows* se ha podido encontrar uno que hacía que las comprobaciones casi siempre diesen negativo: el fichero `Thumbs.db` [21]. Este fichero oculto, que se crea automáticamente, se encarga de guardar las miniaturas que se visualizan mediante vista previa al navegar por el explorador de archivos. El hecho de que aparezca este fichero nuevo podría hacer que el *test* fallara, ya que siempre daría como mínimo que ha aparecido un fichero nuevo. Aunque *Windows* permite desactivar la creación de estos ficheros, como esta operación la debería llevar a cabo cada usuario para que el *myum* funcionara correctamente, para evitar que el programa sea dependiente de otras configuraciones del usuario, se ha optado por mantener una lista de ficheros a omitir, que de momento contiene dicho fichero, y en caso de que la diferencia de ficheros la dé la aparición de un fichero de esa lista, se omite y no pasa a la lista final de ficheros añadidos. En principio la lista solo dispone del fichero que se ha tratado, aunque si en un futuro apareciese algún otro fichero similar o en una nueva versión de *Windows* futura se renombrase, se podría cambiar el nombre o añadir un nuevo elemento a la lista fácilmente.

Una vez realizadas todas las comprobaciones anteriores, si no hay ficheros eliminados, añadidos o modificados, el *test* se dará por correcto y en caso negativo, se indicarán los errores producidos. Si se utiliza la opción de comprobar paquete, un resultado negativo simplemente será información para el usuario, pero en otras operaciones de instalación o desinstalación puede implicar la no realización de la operación.

6.1.1. Ejemplos de comprobación de paquetes

Para concluir la sección de la comprobación de la integridad se mostrarán dos ejemplos prácticos de esta funcionalidad en el que hay resultados dispares. Para estos ejemplos se supone instalado el paquete *basic*.

El primer ejemplo (figura 34) muestra la salida por pantalla del programa de lo que ocurriría nada más instalar un paquete en el que no hay ningún cambio y la comprobación es correcta. Para el segundo ejemplo (figura 35) se han modificado dos ficheros, se ha añadido uno nuevo y se ha eliminado otro, por lo que aparecen cuatro `WARNING`, informando de los cuatro

```
Pedro@Pedro-HP /cygdrive/d/myum/pmoreno/src
$ python3 myum.py -m mirror_list_cygwin.json -ck basic
myum INFO: Workspace is /cygdrive/d/myum/pmoreno
myum INFO: Repository mirror list file is mirror_list_cygwin.json
myum INFO: Checking package basic integrity...
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic.doxygen OK.
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic_doxygen.bat OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin.svg OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.btim OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.svg OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo_rtl.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/basic_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_auto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_nauto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc16_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc32_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc8_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/ddrin.vhd OK.
myum INFO: File design/devices/source/crip_basic/hds/sfiforegrst/symbol.sb OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/_rtl.bd._fpf OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/rtl.bd OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/symbol.sb OK.
myum INFO: File design/devices/source/crip_basic/vsim/files0.txt OK.
myum INFO: File design/devices/source/crip_basic/vsim/modelsim.ini OK.
myum INFO: File design/devices/source/crip_basic/vsim/vcompiler.bat OK.
myum INFO: File design/devices/source/crip_basic/vsim/vcompiler.sh OK.
myum INFO: Package basic version 1.0.1 integrity check is OK, no files have been changed since last installation.
```

Figura 34: Comprobación de un paquete correcta.

problemas encontrados durante la comprobación del paquete.

6.2. Forzar la instalación

Otra de las funcionalidades del programa, que ya se adelantó cuando se explicó la instalación de paquetes es la de forzar la instalación. Cuando se realiza una instalación, se realizan una serie de comprobaciones y en caso de que el resultado de las mismas sea negativo, se detiene la instalación. La opción de forzar sirve para que el paquete continúe instalándose a pesar de que de que las comprobaciones no hayan dado un resultado correcto. Por ello, aunque esta función es muy útil, debe usarse con mucho cuidado y no debe usarse a la primera aparezca un error porque se podrían eliminar datos no deseados o establecer una configuración que no ofrezca los requisitos que el usuario quiere. Del mismo modo, en las operaciones de desinstalar, instalación específica, actualización o borrado automático esta opción se utiliza para proseguir dichos procesos en caso de que alguna comprobación no tenga éxito. En la tabla 5 se muestran las comprobaciones que esta opción permite que el programa continúe (y esto no significa que las comprobaciones se dejen de hacer, sino que el *myum* sigue realizando las operaciones independientemente del resultado) para las distintas opciones que permiten forzado.

6.3. Desinstalación de paquetes

En el capítulo anterior se explicó el proceso de instalación de paquetes, pero una vez completado debe poder ser reversible. Para poder eliminar la instalación, existe la opción de desinstalar (`-u` o `--uninstall`), que afortunadamente es mucho más sencilla que la operación contraria. Tan simple es que se puede resumir en cuatro pasos:

1. Comprobación de la integridad del paquete (que incluye la comprobación de que el paquete

```
Pedro@Pedro-HP /cygdrive/d/myum/pmoreno/src
$ python3 myum.py -m mirror_list_cygwin.json -ck basic
myum INFO: Workspace is /cygdrive/d/myum/pmoreno
myum INFO: Repository mirror list file is mirror_list_cygwin.json
myum INFO: Checking package basic integrity...
myum WARNING: File design/devices/source/crip_basic/vsim/vcompiler.sh has been deleted
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic.doxygen OK.
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic_doxygen.bat OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin.svg OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.btim OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.svg OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo_rtl.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/basic_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_auto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_nauto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc16_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc32_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc8_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/ddrin.vhd OK.
myum INFO: File design/devices/source/crip_basic/hds/sfiforegrst/symbol.sb OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/_rtl.bd._fpf OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/rtl.bd OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/symbol.sb OK.
myum WARNING: File design/devices/source/crip_basic/vsim/files0.txt has changed.
myum INFO: File design/devices/source/crip_basic/vsim/modelsim.ini OK.
myum WARNING: File design/devices/source/crip_basic/vsim/vcompiler.bat has changed.
myum WARNING: File: design/devices/source/crip_basic/vsim/new_file_to_test.txt has been added after installing the pac
kage.
myum INFO: There are files that have changed since installation. Revise them.
```

Figura 35: Comprobación de un paquete en el que hay ficheros modificados, añadidos y eliminados.

Comprobación	Instalación (mediante fichero o específica)	Actualización	Desinstalación	Borrado automático
Comprobación de si un repositorio está limpio.	X	X	-	-
Comprobación de la versión instalada del paquete que se solicita instalar para evitar instalarlo si no es a una versión más alta compatible.	X	X	-	-
Comprobación de la integridad del paquete.	X	X	X	X

Tabla 5: Comprobaciones cuyo resultado puede ser omitido mediante la opción de forzado.

esté efectivamente instalado).

2. Obtención de la lista de ficheros y directorios instalados del fichero de datos del paquete (`paquete.json`).
3. Borrar los ficheros y los directorios recursivamente.
4. Actualizar el fichero de datos de la instalación.

La primera parte del proceso es realizar la comprobación de la integridad, del modo explicado con anterioridad. Si no hay ficheros modificados, añadidos o eliminados entonces la desinstalación se podrá llevar a cabo; en caso contrario, solo se efectuará si se utiliza la opción de forzado. Posteriormente se lee la clave `'data'` del fichero de datos del paquete, que indica los directorios que se copiaron del repositorio y los ficheros individuales que se instalaron. Para el caso de los ficheros, se los eliminará sin más, y para el caso de los directorios se utilizará una función de borrado recursivo que borra la carpeta indicada, eliminando el contenido de todos los ficheros y subdirectorios que se encuentren dentro de ella. Cuando estos tres pasos queden realizados, ya no quedará ningún dato del paquete en el directorio de trabajo.

Para terminar la desinstalación, es necesario modificar el fichero de datos de instalación (`.myum.db`). En la parte de paquetes instalados (`'installed'`) hay que eliminar la entrada en la que aparecía el paquete instalado y, por otra parte, en el historial, hay que añadir un nuevo registro en el cual aparecerán el nombre del paquete, versión, tipo de paquete (solicitado o dependencia), hora de desinstalación y tipo de operación, que en este caso es desinstalación (`'uninstall'`).

Para terminar de comprender el proceso, se mostrará como ejemplo la desinstalación del paquete *basic*, que se mostró como ejemplo para la comprobación de la integridad. En primera instancia, como en el último ejemplo había cambios en el paquete, aparecerá un error y la instalación no será posible (figura 36). Para solucionar este problema se utiliza la opción `-f` de forzado (que se muestra en el comando inicial), tras la cual la desinstalación es correcta (figura 37), aunque se muestren las advertencias correspondientes por los problemas de integridad.

6.4. Listar los paquetes instalados

Una de las opciones interesantes de *myum* es la que te da la posibilidad de mostrar los datos de los paquetes instalados sin necesidad de acudir al fichero de datos sobre la instalación (`.myum.db`), que aparte de que el usuario no tiene necesidad de conocer su existencia, al contener información de paquetes instalados, historial y la lista de todos los ficheros instalados, puede resultar más incómodo para su lectura. Por ello, *myum* mediante la opción `-li` (o `--list_installed`) permite mostrar un resumen en el que aparecen el nombre del paquete, la versión, el tipo (si fue solicitado o fue una dependencia) y la hora de instalación. La figura 38 muestra un ejemplo claro de esta funcionalidad. Al inicio, al no haber ningún paquete instalado se indica que no hay paquetes. Posteriormente se instala el paquete *apb* 4.0.0 que tiene como dependencia el paquete *basic 4.0.0* (para focalizarse solo en la funcionalidad de esta sección, se

```
Pedro@Pedro-HP /cygdrive/d/myum/pmoreno/src
$ python3 myum.py -m mirror_list_cygwin.json -u basic
myum INFO: Workspace is /cygdrive/d/myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is mirror_list_cygwin.json
myum INFO: Checking package basic integrity...
myum WARNING: File design/devices/source/crip_basic/vsim/vcompiler.sh has been deleted
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic.doxygen OK.
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic_doxygen.bat OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin.svg OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.btim OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.svg OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo_rtl.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/basic_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_auto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_nauto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc16_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc32_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc8_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/ddrin.vhd OK.
myum INFO: File design/devices/source/crip_basic/hds/sfiforegrst/symbol.sb OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/_rtl.bd._fpf OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/rtl.bd OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/symbol.sb OK.
myum WARNING: File design/devices/source/crip_basic/vsim/files0.txt has changed.
myum INFO: File design/devices/source/crip_basic/vsim/modelsim.ini OK.
myum WARNING: File design/devices/source/crip_basic/vsim/vcompiler.bat has changed.
myum WARNING: File: design/devices/source/crip_basic/vsim/new_file_to_test.txt has been added after installing the pac
kage.
myum ERROR: There are changes in your installed version respect to the original version. Check them and or if you real
ly want to uninstall the package and lose changes, you can use -f (--force) option.
```

Figura 36: Desinstalación errónea al fallar la comprobación de la integridad.

```
Pedro@Pedro-HP /cygdrive/d/myum/pmoreno/src
$ python3 myum.py -m mirror_list_cygwin.json -u basic -f
myum INFO: Workspace is /cygdrive/d/myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is mirror_list_cygwin.json
myum INFO: Checking package basic integrity...
myum WARNING: File design/devices/source/crip_basic/vsim/vcompiler.sh has been deleted
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic.doxygen OK.
myum INFO: File design/devices/source/crip_basic/doxygen/crip_basic_doxygen.bat OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin.svg OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.btim OK.
myum INFO: File design/devices/source/crip_basic/doxygen/ddrin_wave.svg OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/afifo_rtl.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/basic_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_auto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/clksel_rtl_nauto.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc16_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc32_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/crc8_pkg.vhd OK.
myum INFO: File design/devices/source/crip_basic/hdl/ddrin.vhd OK.
myum INFO: File design/devices/source/crip_basic/hds/sfiforegrst/symbol.sb OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/_rtl.bd._fpf OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/rtl.bd OK.
myum INFO: File design/devices/source/crip_basic/hds/sync/symbol.sb OK.
myum WARNING: File design/devices/source/crip_basic/vsim/files0.txt has changed.
myum INFO: File design/devices/source/crip_basic/vsim/modelsim.ini OK.
myum WARNING: File design/devices/source/crip_basic/vsim/vcompiler.bat has changed.
myum WARNING: File: design/devices/source/crip_basic/vsim/new_file_to_test.txt has been added after installing the pac
kage.
myum WARNING: There are changes in your installed version respect to the original version.
myum INFO: Package basic version 1.0.1 has been successfully uninstalled
```

Figura 37: Desinstalación correcta utilizando la opción de forzado.

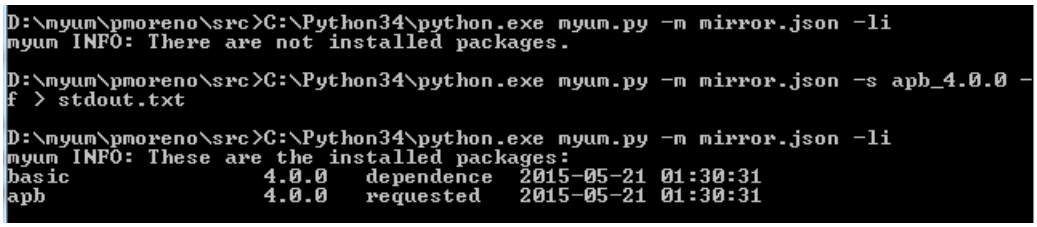

Figura 38: Listado de paquetes antes y después de una instalación.

ha redirigido la salida del comando de instalación a un fichero) y por último se vuelve a pedir la lista de paquetes instalados donde ahora aparecen los dos paquetes por orden de instalación (siempre en una instalación se instalan las dependencias antes que los paquetes, puesto que para el funcionamiento del último se requiere la instalación del primero, lo cual el algoritmo recursivo de generación del árbol de dependencias lo consigue directamente) junto con el detalle de la instalación.

6.5. Instalación específica

En el ejemplo de la sección anterior (figura 38), cuando se instaló el paquete *apb*, no se utilizó el fichero de datos de la instalación, sino que se pidió la instalación del paquete de forma manual. En ocasiones, puede ser más rápido utilizar directamente la línea de comandos para instalar un paquete que tener que abrir el fichero e incluir el paquete. Por ello, existe la posibilidad de realizar la instalación específica de un único paquete mediante la línea de comandos. Esta opción es la `-s` (o `--specific`). Para poderla usar hay que poner el nombre de la opción de una de las dos formas seguido del paquete que se quiera instalar en el formato `paquete_x.y.z` donde `x.y.z` son los tres componentes de la versión. En este caso, al pedir de forma específica un paquete, no es posible indicar que solo se quieren las dependencias con el `d.d.d`. A partir de la entrada por línea de comandos, se separa el nombre de la versión y se realiza el proceso de instalación como si se tratara de un fichero de instalación con únicamente un paquete. En el último paso de la instalación, como se actualiza el fichero de instalación, sin necesidad de haberlo tenido que editar, se incluirá el paquete solicitado por si en el futuro se quiere realizar una misma instalación, que quede constancia de este paquete que no se ha instalado por el método estándar. Como ejemplo de esta opción está la instalación de la figura 38. En el comando adicionalmente se ha especificado un fichero de instalación con la opción `-m` y se ha utilizado la opción de forzado con `-f`, aparte de la redirección de la salida al fichero `stdout.txt`, aunque si no se desea el efecto de estas últimas opciones, se podría utilizar esta opción de forma individual. La salida que quedó almacenada en el fichero es muy similar a la de la instalación estándar:

```
myum INFO: Workspace is D:\myum\pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is mirror.json
myum INFO: The following packages were required to install:
1    apb                 4.0.0     (requested)

myum INFO: The following packages will be installed after solving dependencies:
1    basic               4.0.0     (dependence)
2    apb                 4.0.0     (requested)
```

```
myum INFO: Installing packages...
myum INFO: Installing package basic version 4.0.0 ...
myum INFO: Executing commands for package basic version 4.0.0 ...
myum INFO: Command echo Test execution was successfully executed.
myum INFO: Package basic version 4.0.0 was successfully installed.
myum INFO: Installing package apb version 4.0.0 ...
myum INFO: Executing commands for package apb version 4.0.0 ...
myum INFO: Package apb version 4.0.0 was successfully installed.
myum INFO: Installation completed!
myum INFO: Rewritting install definition file...
myum INFO: Finished!!
```

6.6. Modo de instalación no automático

La siguiente de las opciones tiene que ver con el comportamiento durante la resolución de dependencias. Por defecto, cuando existe un conflicto o se presentan varias alternativas de posibles versiones para instalar de un paquete, el programa buscará la versión más alta compatible en caso de que las versiones sean compatibles o la versión que se utilizó la última vez cuando se produjo el mismo conflicto entre versiones incompatibles con el mismo paquete, si es que la misma situación se produjo con anterioridad. Lo que añade esta opción (`-na`, o `--noautomatic`) es la posibilidad de que el usuario tenga mayor control y pueda tomar las decisiones entre qué versión elegir en vez de dejar al *myum* que decida por sí solo. Siempre que exista más de una alternativa, se mostrará el diálogo en el que se presentarán todas las opciones disponibles, junto con la opción de abortar, por si se prefiere consultar información sobre los paquetes antes de realizar la instalación definitiva o si se prefiere buscar otras soluciones; y la opción de omitir el paquete. Los casos en los que se preguntará al usuario en lugar de utilizar la resolución automática son los siguientes:

- Cuando solo haya una versión para instalar de un paquete tras resolver el árbol de dependencias, pero exista una versión más alta compatible para ese paquete.
- Cuando en el árbol de dependencias existan varias versiones compatibles del mismo paquete, en cuyo caso se mostrarán dichas versiones y la más alta compatible si no estuviera entre ellas.
- Cuando existan varias versiones incompatibles para instalar para un paquete, en cuyo caso se mostrarán dichas versiones, junto con la más alta compatible para cada una de las versiones (si es la misma para varias, solo se mostrará una vez).

6.7. Limpieza del directorio caché

Cuando *myum* realiza la mayoría de sus operaciones, necesita obtener algunos datos temporales. Por ejemplo, cuando necesita el fichero de datos del paquete (`paquete.json`), necesita descargarlo del repositorio y almacenarlo en el directorio caché de forma provisional. Del mismo modo, cuando se descarga el contenido del paquete, el fichero comprimido (*tarball*) se

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -cl
myum INFO: Workspace is /home/projects/cr_tools_myum/pmoreno
myum INFO: Cache directory has been cleaned.
```

Figura 39: Limpieza del directorio caché.

deja también en este directorio. El contenido, una vez procesado, no tiene ninguna importancia y son solo restos que deja el programa, por lo que pueden ser eliminados. La opción `-cl` (o `--clean`) permite eliminar todo el contenido del directorio caché para no tener almacenados datos en el equipo que en realidad no se están utilizando. Cuando se ejecuta, como salida muestra el directorio de trabajo, a partir del cual estará el directorio caché bajo `/work/myum` y un mensaje indicando si la operación se ha realizado o no con éxito, tal y como se muestra en la figura 39.

6.8. Verbosidad del programa

Durante la ejecución del programa aparecen diferentes mensajes por pantalla para notificar al usuario sobre las distintas acciones llevadas a cabo o los posibles problemas que ocurran durante la instalación. La abundancia o ausencia de dichos mensajes es otro aspecto que se puede controlar y para ello se presentarán un par de opciones: la opción `-nv` (o `--noverbose`) y `-db` (`--debug`).

Para realizar el sistema de control de mensajes se utiliza el módulo `logging` de Python [14], por el cual se crea un objeto que será el encargado de mostrar los mensajes. Para el reporte de mensajes se definen varios niveles: `CRITICAL`, `ERROR`, `WARNING`, `INFO` y `DEBUG`. Cada uno de estos niveles permite mostrar los mensajes generados por su nivel y el de los anteriores (ej. en el nivel `ERROR`, se mostrarían los mensajes `CRITICAL` y `ERROR`; mientras que en el `INFO`, se mostrarían `CRITICAL`, `ERROR`, `WARNING` e `INFO`). En el programa se ha decidido hacer uso de todos los niveles excepto del primero y su uso particular de cada uno de ellos es el siguiente:

- `ERROR`: Muestra aquellos mensajes de error que impiden que la operación pueda continuar.
- `WARNING`: Muestra aquellas advertencias que el programa reporta cuando ocurre algún problema que el programa puede manejar u omitir, de modo que la operación en curso continúa.
- `INFO`: Muestra mensajes de información sobre las tareas que se están llevando a cabo. Pueden ser tanto mensajes informativos (del tipo 'Instalando paquete X...'), como mensajes de éxito (del tipo 'La operación se ha realizado con satisfactoriamente').
- `DEBUG`: Muestra mensajes externos al programa que se producen mientras se ejecutan comandos al final de la instalación. Al no ser producidos por *myum*, sino por el comando que se mande ejecutar, se los separa del resto.

Por defecto, el programa muestra los mensajes de los tres primeros grupos (error, advertencia e información), mientras que los mensajes de comandos externos en principio no son mostrados. El color elegido para mostrar los mensajes en *Linux* y *Cygwin* para los cuatro grupos es rojo, naranja, verde y azul, siguiendo el orden de la lista. Para cambiar este comportamiento existen las siguientes opciones:

- Modo no verboso: Únicamente muestra los mensajes que tiene un resultado negativo para el programa, es decir, los errores y las advertencias. Este modo se activa con la opción `-nv` (o `--noverbose`).
- Modo *debug*: Muestra todos los mensajes incluidos aquellos que generan otros programas mediante la ejecución de comandos externos. Este modo se activa con la opcíon `-db` (o `--debug`).

6.8.1. Historial de mensajes (`myum.log`)

Adicionalmente al propio reporte de mensajes en la línea de comandos también puede convenir almacenar el resultado de la ejecución de los comandos en un fichero. Por ello, dado que el objeto que muestra los mensajes permite añadir un manejador de ficheros, se mantiene un registro de todos los mensajes en el fichero `myum.log`, que se encuentra en el directorio `tools/myum` dentro del directorio de trabajo.

La diferencia de este fichero con respecto a los mensajes que se muestran por la consola es que en este caso se ha decidido no hacer distinciones entre niveles de verbosidad. La razón es que puede ser que un usuario podría desear no recibir mucha información sobre una operación, pero cuando recibe un error le hubiese gustado haber visto todas operaciones paso a paso, no solo el mensaje de error como había pedido. En ese caso, si en el fichero se almacenan todos los mensajes, aunque en la pantalla solo viese parte de los mismos, podría completar su información leyendo el fichero.

6.9. Actualización de paquetes

Mediante la opción de instalar, se presentó que se podía instalar una serie de paquetes y dependencias en las que de forma automática se instalaba la versión más alta compatible. Pero, ¿qué ocurre si posteriormente a la instalación aparece una nueva versión? En dicho caso, el usuario tendría una versión que no será la más alta compatible, pero le podría interesar realizar una actualización. Para poder realizar estas actualizaciones surge la opción `-up` (o `--update`).

Esta opción mira para cada uno de los paquetes instalados si existe una versión más alta compatible (con el primer dígito igual) y se encarga de mostrar una lista de todos los paquetes que pueden ser actualizados sin tener problemas de compatibilidad. Este hecho es de vital importancia porque lo que se desea garantizar con una actualización en este modo es que aunque se cambie la versión, siga funcionando todo lo que hay, y simplemente se añadan mejoras que puedan corregir errores o dotar de mayor funcionalidad.

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -up
myum INFO: Workspace is /home/projects/cr_tools_myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is /home/eda/myum/myum_mirror_list.json
myum INFO: The following packages can be updated:
arithmetic          1.0.0               1.0.1               dependence

myum ERROR: Working repository has uncommitted changes and cannot install. Commit all your changes or use -f to force installation.
```

Figura 40: Actualización de paquetes errónea por falta de repositorio limpio.

Una vez que se elabora la lista, se muestra al usuario y este tiene la opción de elegir si lleva a cabo todas las actualizaciones o no. En el caso de que quisiera llevar a cabo solo una de ellas, esta opción no lo permitiría, pero al ser una versión más alta compatible podría hacerlo sin ningún problema con la opción de instalación específica, y como en la lista se mostraría la versión más alta, no habría dificultad en realizar la actualización mediante el otro método.

Si finalmente el usuario se decide por realizar la actualización, se incorporaran los paquetes a una lista del formato del fichero de instalación y se procederá a utilizar las funciones de instalación, con la única diferencia de que el tipo de paquete (solicitado o dependencia) se mantendrá, en lugar de poner a todos `'requested'` por el hecho de que se solicita la actualización. La razón es que si una dependencia se actualiza, de este modo se puede saber siempre que ese paquete es una dependencia y poder tratarla como tal durante el desarrollo del proyecto.

Durante esta operación se realizarán las mismas comprobaciones que para la instalación, por lo que también es posible omitir el resultado de algunas con la opción de forzado. Estas comprobaciones son las de si el repositorio está limpio, la de que en realidad la versión sea la más alta compatible (en realidad esta comprobación nunca va a dar negativo porque los paquetes que se van a actualizar lo van a hacer a una versión más alta compatible de antemano, pero al usar el motor de instalación, la comprobación se realiza) y la comprobación de la integridad. Para poner ejemplo de esta funcionalidad, la figura 40 muestra una actualización que no pudo ser llevada a cabo por no tener el repositorio limpio y la actualización final realizada con la opción de forzado (figura 41).

6.10. Borrado automático

Muchos de los paquetes instalados no se instalan por voluntad del usuario, sino porque son dependencias de otros paquetes. Cuando se produce una desinstalación de un paquete, únicamente se elimina el paquete en cuestión y podría ocurrir que hubiese dependencias que se instalaron para hacer funcionar un paquete, pero que ya no se estén utilizando para nada, puesto que el paquete que obligó a instalarlas ya esté desinstalado. Para eliminar esas dependencias que ya no son útiles existe la opción de autoborrado `-ar` (o `--autoremove`).

Para realizar esta operación, se lee la lista de paquetes instalados y el tipo (solicitado o dependencia) y se analiza si aquellos que tienen como tipo dependencia realmente son dependencia de algún paquete solicitado y en caso negativo podrían ser eliminados salvo una excepción principal, la de los paquetes que se están desarrollando. Si se está desarrollando un

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -up -f
myum INFO: Workspace is /home/projects/cr_tools_myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is /home/eda/myum/myum_mirror_list.json
myum INFO: The following packages can be updated:
arithmetic          1.0.0               1.0.1               dependence

myum WARNING: Working repository has uncommitted changes.
Do you want to continue? (y/n)
y
myum INFO: Checking package arithmetic integrity...
myum INFO: File design/devices/source/crip_arithmetic/vsim/modelsim.ini OK.
myum INFO: File design/devices/source/crip_arithmetic/vsim/vcompiler.bat OK.
myum INFO: File design/devices/source/crip_arithmetic/vsim/vcompiler.sh OK.
myum INFO: File design/devices/script/hds/crip_arithmetic.src.hdp OK.
myum INFO: Package arithmetic is installed in version 1.0.0 but according to your query, it will be changed to version 1.0.1.
myum INFO: Updating packages...
myum INFO: Installing package arithmetic version 1.0.1 ...
myum INFO: Executing commands for package arithmetic version 1.0.1 ...
myum INFO: Command python3 tools/myum/arithmetic_post_install.py execution was successfully executed.
myum INFO: Package arithmetic version 1.0.1 was successfully installed.
myum INFO: All packages have been updated to their highest compatible version!
```

Figura 41: Actualización de paquetes mediante la opción `-up`.

paquete, sus dependencias aparecerán con el tipo `'dependence'`, pero como el paquete que se desarrolla (que aparece en el fichero de instalación con versión `d.d.d`) no está en la lista de instalados, se añade un mecanismo para que estas dependencias nunca puedan ser autoeliminadas. A modo de ejemplo, la figura 42 muestra un caso en el que dos paquetes *arithmetic* y *registers* dependían del paquete *apb*, pero este se eliminó y al utilizar esta opción, los dos paquetes fueron borrados.

6.11. Historial de instalación

Cuando se han realizado multitud de operaciones con el programa, es posible que se quiera visualizar de algún modo todas las operaciones realizadas. El fichero de datos de instalación (`.myum.db`) almacena el histórico de la instalación, aunque su formato de diccionario y la gran cantidad de datos puede hacer que su lectura no sea muy cómoda para el usuario. Para ello, existe la opción `-hs` (o `--history`), que permite mostrar todas las operaciones registradas. Para cada una de ellas aparece el tipo de operación (instalación, reinstalación, actualización o desinstalación), el nombre del paquete, la versión, el tipo (solicitado o dependencia) y la fecha y hora. La figura 43 muestra un ejemplo de salida que se produce mediante esta opción.

6.12. Modo de simulación

La última de las funcionalidades que se presentan en el programa es la del modo simulación. Es posible que antes de realizar una operación, ya sea una instalación, desinstalación o actualización se desee conocer si el proceso va a funcionar bien o qué conflictos va a haber, qué paquetes se instalarían tras resolver dependencias, si en caso de realizar comprobaciones de integridad qué resultado darían, etc. En dicho caso, existe la opción `-dr` (o `--dry`) que permite simular las operaciones, pero sin hacerlas efectivas. Por ejemplo para el caso de la instalación, se realizaría todo el proceso de resolución de dependencias (e incluso si se resuelve un conflicto

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -ar
myum INFO: Workspace is /home/projects/cr_tools_myum/pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is /home/eda/myum/myum_mirror_list.json
myum INFO: Package registers will be uninstalled.
myum INFO: Checking package registers integrity...
myum INFO: File design/devices/source/crip_registers/doxygen/crip_registers.doxygen OK.
myum INFO: File design/devices/source/crip_registers/doxygen/crip_registers_doxygen.bat OK.
myum INFO: File design/devices/source/crip_registers/hdl/registers_pkg.vhd OK.
myum INFO: File design/devices/source/crip_registers/hds/.hdlsidedata/_registers_pkg.vhd._fpf OK.
myum INFO: File design/devices/source/crip_registers/vsim/files0.txt OK.
myum INFO: File design/devices/source/crip_registers/vsim/modelsim.ini OK.
myum INFO: File design/devices/source/crip_registers/vsim/vcompiler.bat OK.
myum INFO: File design/devices/source/crip_registers/vsim/vcompiler.sh OK.
myum INFO: File design/devices/script/hds/crip_registers.src.hdp OK.
myum INFO: File tools/myum/registers_post_install.py OK.
myum INFO: Package registers version 1.0.0 has been successfully uninstalled
myum INFO: Package arithmetic will be uninstalled.
myum INFO: Checking package arithmetic integrity...
myum INFO: File design/devices/source/crip_arithmetic/hds/sat_c2_2s/symbol.sb OK.
myum INFO: File design/devices/source/crip_arithmetic/hds/square_c2/rtl.bd OK.
myum INFO: File design/devices/source/crip_arithmetic/hds/square_c2/symbol.sb OK.
myum INFO: File design/devices/source/crip_arithmetic/hds/sub_c2/rtl.bd OK.
myum INFO: File design/devices/source/crip_arithmetic/hds/sub_c2/symbol.sb OK.
myum INFO: File design/devices/source/crip_arithmetic/vsim/files0.txt OK.
myum INFO: File design/devices/source/crip_arithmetic/vsim/modelsim.ini OK.
myum INFO: File design/devices/source/crip_arithmetic/vsim/vcompiler.bat OK.
myum INFO: File design/devices/source/crip_arithmetic/vsim/vcompiler.sh OK.
myum INFO: File design/devices/script/hds/crip_arithmetic.src.hdp OK.
myum INFO: File tools/myum/arithmetic_post_install.py OK.
myum INFO: Package arithmetic version 1.0.1 has been successfully uninstalled
myum INFO: Operation completed!
```

Figura 42: Borrado automático de paquetes

```
pmoreno@simulation2 /home/projects/cr_tools_myum/pmoreno/src (feature/myum_2.0)
$python3 myum.py -hs
myum INFO: These are the operations performed:
install       basic              1.0.1   requested   2015-05-13 14:51:37
install       arithmetic         1.0.0   dependence  2015-05-13 14:51:39
install       driver_common      1.0.0   requested   2015-05-13 14:53:01
reinstall     basic              1.0.1   requested   2015-05-13 14:53:04
reinstall     arithmetic         1.0.0   dependence  2015-05-13 14:53:06
uninstall     basic              1.0.1   requested   2015-05-19 12:06:40
install       basic              1.0.2   requested   2015-05-20 15:07:58
uninstall     driver_common      1.0.0   requested   2015-05-21 11:09:57
install       registers          1.0.0   dependence  2015-05-21 11:11:25
install       obt_setup          1.0.0   dependence  2015-05-21 11:11:28
reinstall     basic              1.0.2   requested   2015-05-21 11:11:31
update        arithmetic         1.0.1   dependence  2015-05-21 11:13:31
uninstall     registers          1.0.0   dependence  2015-05-21 11:14:34
uninstall     arithmetic         1.0.1   dependence  2015-05-21 11:14:37
```

Figura 43: Historial de operaciones

```
D:\myum\pmoreno\src>C:\Python34\python.exe myum.py -m mirror.json -s memory_1.0.
0 -f -dr
myum INFO: Workspace is D:\myum\pmoreno
myum INFO: Definition installation file is myum_install_def.json
myum INFO: Repository mirror list file is mirror.json
myum WARNING: Working repository has uncommitted changes.
myum INFO: The following packages were required to install:
1    memory              1.0.0     (requested)

myum INFO: Package arithmetic will be installed in its highest compatible versio
n 1.1.0.
myum INFO: Package basic will be installed in its highest compatible version 1.0
.1.
myum INFO: The following packages will be installed after solving dependencies:

1    arithmetic          1.1.0     (dependence)
2    basic               1.0.1     (dependence)
3    technology          1.0.0     (dependence)
4    memory              1.0.0     (requested)

myum INFO: Installing packages...
myum INFO: Installing package arithmetic version 1.1.0 ...
myum INFO: Executing commands for package arithmetic version 1.1.0 ...
myum INFO: Package arithmetic version 1.1.0 was successfully installed.
myum INFO: Installing package basic version 1.0.1 ...
myum INFO: Executing commands for package basic version 1.0.1 ...
myum INFO: Package basic version 1.0.1 was successfully installed.
myum INFO: Installing package technology version 1.0.0 ...
myum INFO: Executing commands for package technology version 1.0.0 ...
myum INFO: Package technology version 1.0.0 was successfully installed.
myum INFO: Installing package memory version 1.0.0 ...
myum INFO: Executing commands for package memory version 1.0.0 ...
myum INFO: Package memory version 1.0.0 was successfully installed.
myum INFO: Installation completed!
myum INFO: Rewritting install definition file...
myum INFO: Finished!!
```

Figura 44: Modo simulación del *myum*.

entre versiones incompatibles, el resultado se almacenaría en el fichero de conflictos) y las comprobaciones de versiones e integridad, pero en caso de que hubiera que desinstalar una versión para instalar la nueva, no se modificaría. Posteriormente, en la instalación, no se copiarían ni ficheros, ni ejecutarían comandos, ni se actualizarían los ficheros de datos de instalación y de definición de la instalación. No obstante, se mostrarían los mensajes de los distintos procesos que no se llevan a cabo para simular todos los pasos de la instalación. Para el caso de la actualización, como es un caso particular de la instalación, se omitirían los mismos pasos y para la desinstalación se realizaría el *test* de integridad y tras él, al no borrarse los ficheros aparecería el mensaje de 'El paquete X versión X.Y.Z ha sido desinstalado correctamente'. La figura 44 muestra el proceso de instalación del paquete *memory*, que depende de *basic*, *arithmetic* y *technology* mediante la opción de simulación.

7. Interfaz gráfica

En los capítulos anteriores se ha descrito en general la funcionalidad del programa y se ha explicado los fundamentos necesarios para su uso mediante la línea de comandos. De forma alternativa a la misma existe la interfaz gráfica, que ha sido diseñada de forma que mantenga la mayor consistencia posible con la versión por consola. Para la implementación de la misma se ha utilizado la librería `Tkinter` (o `tkinter` para versiones de *Python* a partir de la 3) [22] En este capítulo se tratarán las distintas posibilidades que ofrece esta interfaz.

Antes de arrancar la interfaz de usuario, mediante consola es posible definir el fichero de instalación y fichero de fuentes para que se tomen directamente al arrancar la aplicación con las mismas opciones que para la línea de comandos, si bien estos parámetros se pueden modificar de forma gráfica a posteriori (y si no se definen se tomarán los mismos valores por defecto que en la versión de consola). La ventaja que supone esto es permitir que la ventana se cargue con todos los datos de los ficheros deseados nada más abrirla. La figura 45 muestra la ayuda previa al arranque de la interfaz gráfica.

La primera vez que se abre la interfaz gráfica se muestra una pantalla similar a la de la figura 46. En esta vista se distinguen varias partes:

- Barra de menús: Desde esta barra se pueden modificar el directorio de trabajo o los ficheros de instalación o fuentes, acceder a las opciones principales del programa u obtener ayuda sobre el mismo.
- Barra de opciones: Permite ejecutar algunas opciones importantes del programa, así como definir el nivel de verbosidad y activar o desactivar las opciones de forzado, resolución automática o el modo de simulación
- Panel principal: Este panel muestra todos los paquetes alfabéticamente junto con la información de instalación. Desde él se puede comprobar la integridad de un paquete instalado o marcar el paquete para realizar alguna operación (instalar/desinstalar/reinstalar/actualizar) con él.
- Ventana de paquetes en desarrollo: Muestra los paquetes que se están desarrollando junto con sus dependencias y permite añadir o eliminar dependencias sobre dichos paquetes sin necesidad de editar los ficheros `paquete.json`.
- Panel de mensajes: Muestra los mensajes que se mostrarían por consola durante la ejecución de las distintas operaciones del programa. También realiza la interacción con el usuario en caso de conflictos.

7.1. Barra de menús

Dentro de los menús, el quizás más interesante y más utilizado sea el menú archivo (*File*), puesto que permite modificar el directorio de trabajo, el fichero de fuentes y el fichero

```
D:\myum\pmoreno\src>C:\Python34\python.exe myum_gui.py -h
usage: myum [-h] [-m MIRRORS_FILE] [-d INSTALL_DEF_FILE]

optional arguments:
  -h, --help            show this help message and exit
  -m MIRRORS_FILE, --mirrors MIRRORS_FILE
                        Change file containing the repositories mirrors list.
                        Default: mirror_list.json.
  -d INSTALL_DEF_FILE, --definition INSTALL_DEF_FILE
                        use a specific install definition file instead of the
                        default (install_def.json).
```

Figura 45: Ayuda previa al arranque de la interfaz gráfica

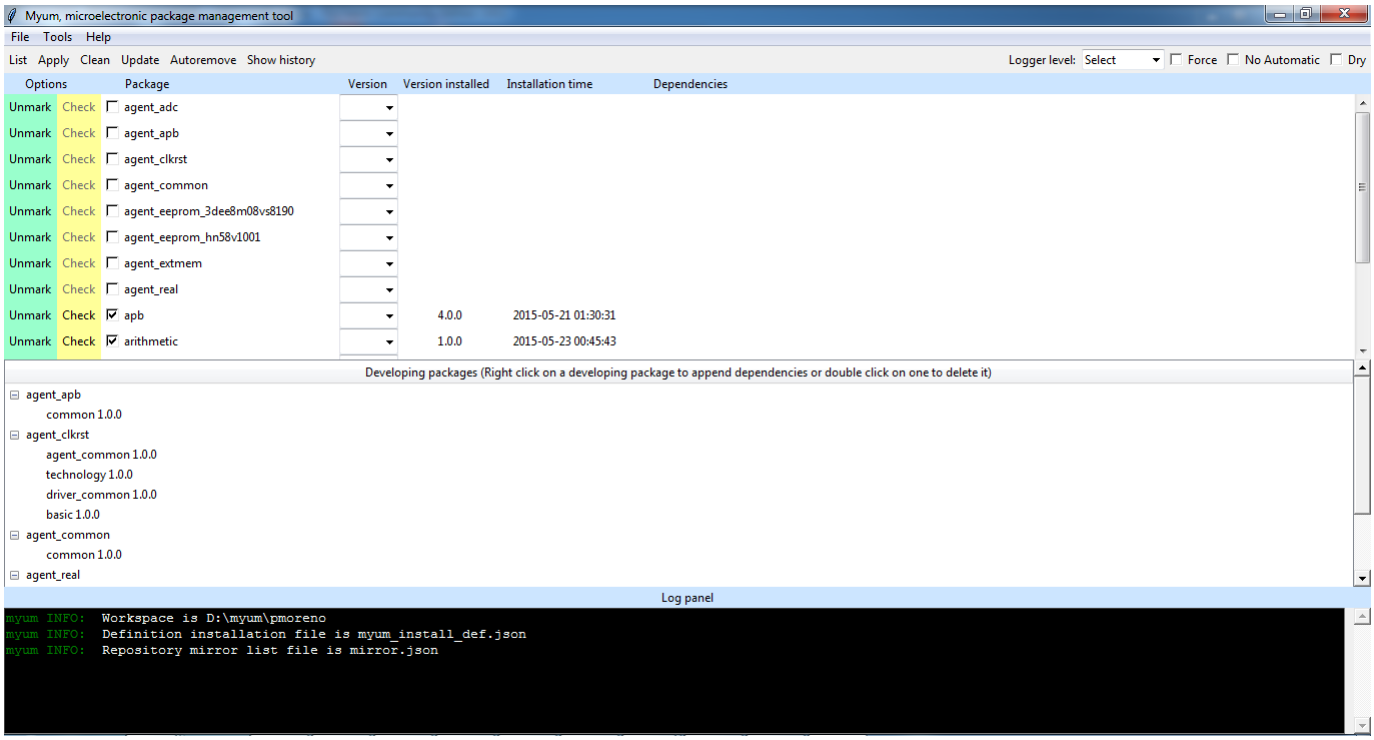

Figura 46: Vista inicial de la interfaz gráfica

de datos de instalación. Dispone de una opción para seleccionar cada uno de estos tres ficheros, que al ser utilizadas abrirán el explorador de archivos del sistema operativo correspondiente y permitirán seleccionar el directorio o fichero que se solicite. Además de estas tres opciones, existe una cuarta para cerrar el programa del mismo modo que puede realizarse mediante la *X* situada en la esquina superior derecha.

El menú de herramientas (*Tools*) proporciona una manera alternativa de acceder a las funciones que permite el programa en la barra de opciones situada debajo de la barra de menús. Consta de distintas opciones para listar paquetes, aplicar operaciones, actualizar paquetes, borrado automático o mostrar el historial. Por último, se puede consultar la ayuda del programa o la versión mediante el menú de ayuda (*Help*).

7.2. Barra de opciones

La barra de opciones se divide en dos partes claramente diferenciadas. La parte de la izquierda en la que se permiten seleccionar las distintas opciones del programa ligadas a operaciones y la parte derecha que permiten activar o desactivar las opciones ligadas a un cambio de comportamiento durante el desarrollo de las operaciones.

En la parte izquierda se muestran las siguientes operaciones:

- Listar: Permite actualizar el panel principal con los paquetes disponibles y la información de instalación.
- Aplicar: Permite instalar, desinstalar, reinstalar o actualizar aquellos paquetes que estén marcados para tal efecto, junto con las dependencias de los paquetes que se estén desarrollando y que aparecen en la ventana de paquetes en desarrollo.
- Limpiar: Borra el directorio caché.
- Actualizar: Actualiza todos los paquetes instalados a su versión más alta compatible.
- Autoborrado: Desinstala aquellos paquetes que son dependencias de paquetes que ya no están instalados.
- Historial: muestra el registro de operaciones en el panel de mensajes con el mismo formato que en la versión de consola.

En la parte derecha se muestran las siguientes opciones:

- Nivel de verbosidad (*Logger Level*): Muestra un desplegable en el cual se puede elegir que el nivel de mensajes sea del tipo `DEBUG` (se muestran todos los mensajes incluyendo los de los comandos externos), `INFO` (muestra errores, advertencias y mensajes de información del propio programa) o `WARNING` (muestra únicamente errores y advertencias). Si no se selecciona nada, por defecto se utilizará el tipo `INFO`.

- Forzado (*Force*): Si se activa el recuadro queda activada la opción de forzado de la instalación.
- No automático *(No Automatic)*: Si se activa el recuadro, la resolución de dependencias no será automática y siempre que exista más de una versión para instalar (incluyendo las recomendadas, es decir, las más altas compatibles), se preguntará al usuario.
- Modo simulación *(Dry)*: Si se activa, se utilizará el modo en el cual las operaciones no quedarán hechas efectivas y simplemente se mostrarían los mensajes producidos durante la ejecución de las operaciones.

7.3. Panel principal

Uno de los elementos claves de la interfaz gráfica es el panel principal, que junta la lista de paquetes disponibles que se mostraba en la línea de comandos con la de paquetes instalados. Previamente se comentó la importancia de tener definidos los ficheros de metadatos antes de ejecutar al inicio y el motivo es que nada más arrancar la aplicación en este panel aparecen todos los paquetes disponibles y los que están instalados de forma alfabética y para ello se necesitan los metadatos. En caso de no estar disponibles, si se seleccionan con el menú Archivo, este panel cambiará automáticamente con los nuevos datos.

Entre los datos que proporciona, tras las opciones que se comentarán posteriormente, aparece el nombre del paquete junto con un recuadro que simplemente indica si el paquete está instalado o no (el usuario no puede controlarlo), un desplegable en el que aparecen todas las versiones disponibles para el paquete, la versión instalada y la fecha de instalación en el caso de que el paquete esté instalado y una columna de dependencias.

La columna de dependencias permite mostrar la lista de las dependencias de un paquete concreto en una versión definida. En principio, como se muestra en la vista inicial, aparece la columna sin datos pero cuando se selecciona una versión en el desplegable de un paquete, cambia y aparecen las dependencias si es que hay. En caso contrario, seguiría apareciendo la fila correspondiente al paquete y versión en blanco.

Entre las opciones de la izquierda, la que se encuentra con fondo amarillo (*Check*) sirve para realizar el *test* de integridad a un paquete instalado. A diferencia de la versión en línea de comandos, esta opción solo está disponible para aquellos paquetes instalados, de modo que se evita el error al solicitar la comprobación en un paquete no instalado. La otra opción está enmarcada dentro del sistema de marcado de paquetes.

7.3.1. Sistema de marcado de paquetes

En la versión del *myum* que utiliza la línea de comandos para realizar una instalación se podría utilizar o bien el fichero de datos de instalación o la instalación específica. La interfaz gráfica permite integrar los dos sistemas en uno solo mucho más potente, que permite realizar más operaciones a la vez.

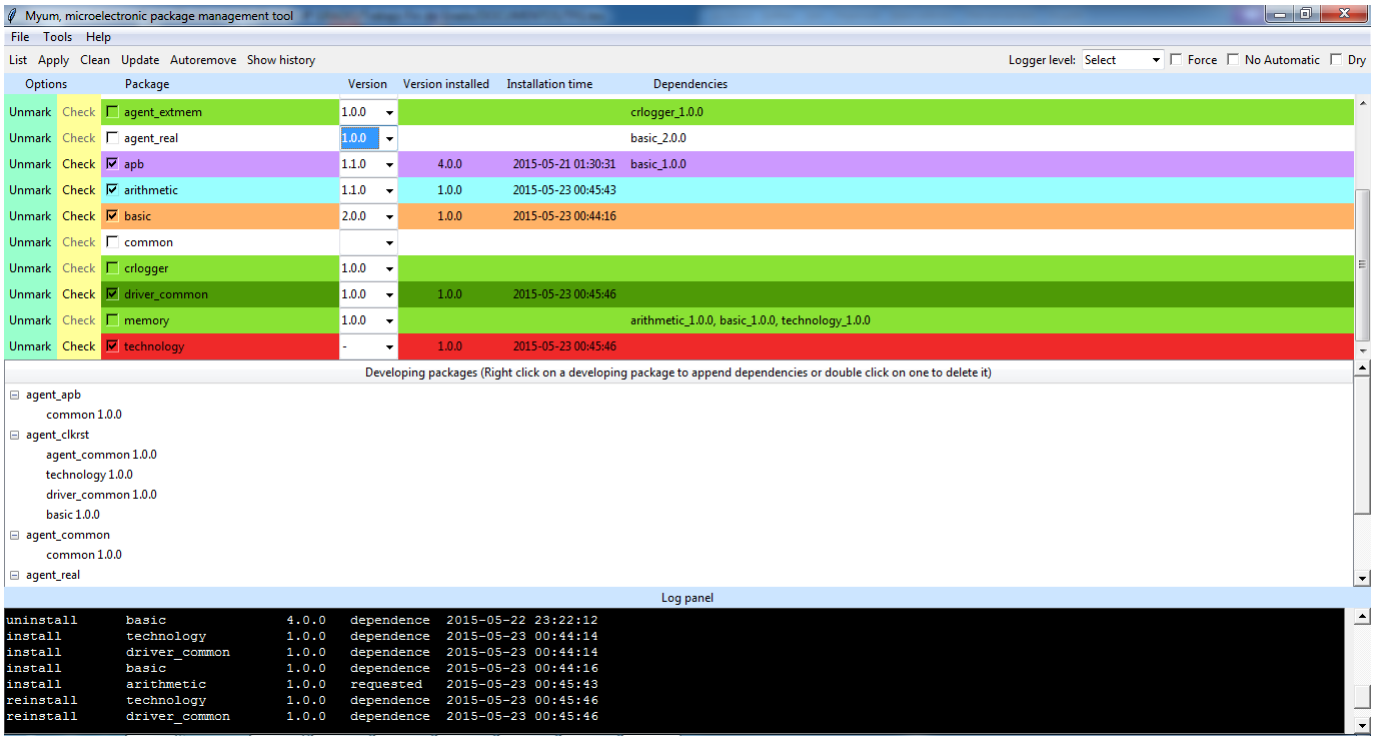

Figura 47: Marcado de operaciones en la interfaz gráfica.

La interfaz mantiene una lista de operaciones que se ejecutarán una vez que se pulse la acción *Aplicar*. Nada más abrir el programa, se lee el fichero de instalación y se añade a dicha lista todos los paquetes solicitados (y para indicar que están en la lista, las filas aparecerán coloreadas). Mediante la interfaz se puede marcar el resto de paquetes para realizar una operación determinada.

Para marcar un paquete simplemente hay que seleccionar una versión en el menú desplegable de opciones. Una vez que quede seleccionado, la fila de dicho paquete quedará coloreada para indicar que el paquete está marcado. El color de la marca dependerá de si el paquete está ya instalado y se actualiza a una versión más alta compatible, incompatible; si es la primera instalación, etc. Dentro de la lista de versiones, aparece una entrada al final en la que aparece un guión (`'-'`). Si el paquete no está instalado, esta opción no tiene efecto, pero si está instalado, el paquete queda marcado para desinstalarse.

Una vez que esté un paquete marcado estaría preparado para realizarse la operación correspondiente, pero en caso de que no se quiera realizar se puede eliminar la marca y eliminar el programa de la lista de operaciones mediante el botón de desmarcar (*Unmark*) que tiene cada paquete a su izquierda con fondo de color verde. Del mismo modo, si un paquete está instalado y en el desplegable se cambia la versión, el paquete pasaría a estar marcado para instalar en la nueva versión que aparezca en el desplegable. La figura 47 muestra una captura en la que aparecen marcados diferentes paquetes para realizar distintas operaciones.

El código de colores utilizado para las distintas marcas es el siguiente:

- Verde: Indica que un paquete está marcado para ser instalado y que actualmente no lo

está.

- Rojo: Indica que un paquete está marcado para ser desinstalado.
- Azul: Indica que un paquete está marcado para ser actualizado a una versión más alta compatible.
- Naranja: Indica que un paquete está marcado para ser actualizado a una versión más alta no compatible. En este caso aunque el paquete esté marcado, si no se utiliza la opción de forzado, al igual que en la línea de comandos, el paquete no se actualizará para mantener consistencia entre las dos versiones.
- Violeta: Indica que un paquete está marcado para ser instalado en una versión anterior a la instalada (compatible o no). Para que la operación se lleve finalmente a cabo se debe utilizar la opción de forzado al igual que en la versión que utiliza la línea de comandos.
- Verde oscuro: Indica que un paquete está marcado para ser reinstalado. La operación únicamente se llevará a cabo si se usa la opción de forzado.

En la figura, en primer lugar aparece el paquete *agent_extmem*, que no está instalado y al haberse seleccionado en versión 1.0.0 se ha marcado para instalar. Al seleccionar la versión, también se ha actualizado el campo de dependencias y aparece que depende del paquete *crlogger* en versión 1.0.0. El segundo paquete, *agent_real*, aparece con la versión seleccionada pero sin marcar. Esto es debido a que se marcó para instalar al seleccionar la versión pero posteriormente se utilizó la opción de desmarcar y no se encuentra entre la lista de paquetes a instalar. El paquete *apb* se encuentra instalado en versión 4.0.0, pero se pide instalar la versión 1.1.0. Como esta versión es anterior, el paquete aparece coloreado en violeta. El paquete *arithmetic* al estar instalado en versión 1.0.0 y solicitarse la 1.1.0, que es más alta y compatible, aparece en azul.

Con respecto al paquete *basic*, está instalado en versión 1.0.0 y como se quiere actualizar a una versión que no es compatible (2.0.0) aparece en naranja. El paquete *common* no tiene seleccionada ninguna versión por lo que no está marcado para instalar. El *crlogger* y el *memory* se encuentran para instalar sin estar instalados (y el *crlogger* no tiene ninguna dependencia al no aparecer en su campo correspondiente); el *driver_common* al solicitarse la instalación en la misma versión en la que está instalado aparece en verde oscuro y por último el paquete *technology*, al seleccionarse como versión el guión, queda marcado para desinstalar y aparece de color rojo.

Al existir una gran cantidad de marcas, al seleccionar la opción de aplicar y llevar a cabo todas las operaciones surge un nuevo conflicto: el caso de que un paquete se marque para instalar y dependa de otro que a su vez esté marcado para desinstalar. En dicho caso se le preguntará al usuario si desea dar prioridad a la instalación, la desinstalación, abortar el proceso o dejar el paquete instalado como se encuentre (lo que habitualmente se ha llamado "omitir el paquete").

En el panel de operaciones se muestra el resultado de todo el proceso. En la figura 46, aparece inicialmente la información de los ficheros de configuración y el directorio de trabajo y en la 47, la salida producida al mostrar el historial. Para concluir el apartado de la instalación,

se mostrará la salida producida al aplicar las operaciones en modo simulación y no automático de las operaciones de la figura 47.

```
myum INFO:  Workspace is D:\myum\pmoreno
myum INFO:  Definition installation file is myum_install_def.json
myum INFO:  Repository mirror list file is mirror.json
myum WARNING: Working repository has uncommitted changes.
myum INFO:  The following packages were required to install:
1    agent_extmem        1.0.0     (requested)
2    apb                 1.1.0     (requested)
3    arithmetic          1.1.0     (requested)
4    basic               2.0.0     (requested)
5    crlogger            1.0.0     (requested)
6    driver_common       1.0.0     (requested)
7    memory              1.0.0     (requested)
8    agent_apb           Only dependencies
9    agent_clkrst        Only dependencies
10   agent_common        Only dependencies
11   common              Only dependencies

myum WARNING: Package technology has been marked to uninstall, but it is also a
    dependence for a package that has been marked to install. Choose one of the
    following actions
1) Unmark uninstallation.
2) Confirm uninstallation.
3) Abort operations.
4) Skip the package.
2
myum INFO:  A higher compatible version is available for package agent_extmem.
Package agent_extmem is required in version 1.0.0
Choose one of the following options:
1) Install package agent_extmem 1.0.0. (requested)
2) Highest compatible version available agent_extmem 1.0.1. (recommended)
3) Abort the program.
4) Skip the package.
1
myum WARNING: Major version conflict with package basic.
Package basic is required in version 1.0.0
Package basic is required in version 2.0.0
Choose one of the following options:
1) Install package basic 1.0.0. (dependence)
2) Install package basic 2.0.0. (dependence)
3) Highest compatible version available basic 1.0.1. (recommended)
4) Abort the program.
5) Skip the package.
5
myum WARNING: Major version conflict with package arithmetic.
Package arithmetic is required in version 1.1.0
Package arithmetic is required in version 1.0.0
Choose one of the following options:
1) Install package arithmetic 1.1.0. (requested)
2) Install package arithmetic 1.0.0. (dependence)
3) Abort the program.
4) Skip the package.
1
myum WARNING: Package apb is already installed in version 4.0.0,  which is
    incompatible with version required 1.1.0 .
```

```
myum WARNING: Package apb is already installed in version 4.0.0 and version
    required 1.1.0 is older.
myum INFO:  Checking package apb integrity...
myum INFO:  File design/devices/source/crip_apb/configuration.xlsx OK.
myum INFO:  File design/devices/source/crip_apb/hdl/apb_pkg.vhd OK.
myum INFO:  File design/devices/source/crip_apb/hdl/apbarbiter.vhd OK.
myum INFO:  File design/devices/source/crip_apb/vsim/vcompiler.sh OK.
myum INFO:  Package apb is installed in version 4.0.0 but according to your
    query, it will be changed to version 1.1.0.
myum INFO:  Checking package arithmetic integrity...
myum INFO:  File design/devices/source/crip_arithmetic/hdl/acc_c2.vhd OK.
myum INFO:  Package arithmetic is installed in version 1.0.0 but according to
    your query, it will be changed to version 1.1.0.
myum INFO:  Package driver_common version 1.0.0 is already installed.
myum INFO:  Checking package driver_common integrity...
myum INFO:  File setup/sw/source/tcl/driver_common/gui.tcl OK.
myum INFO:  File setup/sw/source/tcl/driver_common/hw_test_launcher.tcl OK.
myum INFO:  File setup/sw/source/tcl/driver_common/pkgIndex.tcl OK.
myum INFO:  Package driver_common version 1.0.0 will be reinstalled.
myum INFO:  The following packages will be installed after solving dependencies:
1     crlogger            1.0.0     (requested)
2     agent_extmem        1.0.0     (requested)
3     apb                 1.1.0     (requested)
4     arithmetic          1.1.0     (requested)
5     driver_common       1.0.0     (dependence)
6     memory              1.0.0     (requested)

myum INFO:  Installing packages...
myum INFO:  Installing package crlogger version 1.0.0 ...
myum INFO:  Executing commands for package crlogger version 1.0.0 ...
myum INFO:  Package crlogger version 1.0.0 was successfully installed.
myum INFO:  Installing package agent_extmem version 1.0.0 ...
myum INFO:  Executing commands for package agent_extmem version 1.0.0 ...
myum INFO:  Package agent_extmem version 1.0.0 was successfully installed.
myum INFO:  Installing package apb version 1.1.0 ...
myum INFO:  Executing commands for package apb version 1.1.0 ...
myum INFO:  Package apb version 1.1.0 was successfully installed.
myum INFO:  Installing package arithmetic version 1.1.0 ...
myum INFO:  Executing commands for package arithmetic version 1.1.0 ...
myum INFO:  Package arithmetic version 1.1.0 was successfully installed.
myum INFO:  Installing package driver_common version 1.0.0 ...
myum INFO:  Executing commands for package driver_common version 1.0.0 ...
myum INFO:  Package driver_common version 1.0.0 was successfully installed.
myum INFO:  Installing package memory version 1.0.0 ...
myum INFO:  Executing commands for package memory version 1.0.0 ...
myum INFO:  Package memory version 1.0.0 was successfully installed.
myum INFO:  Installation completed!
myum INFO:  Uninstalling packages ...
myum INFO:  Uninstalling package technology ...
myum INFO:  Checking package technology integrity...
myum INFO:  File design/devices/source/crip_technology/hds/pad_zt5050n4n/symbol.
    sb OK.
myum INFO:  File design/devices/source/crip_technology/vsim/files0.txt OK.
myum INFO:  File design/devices/source/crip_technology/vsim/modelsim.ini OK.
myum INFO:  File design/devices/source/crip_technology/vsim/vcompiler.bat OK.
myum INFO:  File design/devices/source/crip_technology/vsim/vcompiler.sh OK.
```

```
myum INFO:  Package: technology version 1.0.0 has been successfully uninstalled
myum INFO:  Uninstallation completed
myum INFO:  Applied changes finished!
myum INFO:  Rewritting install definition file...
myum INFO:  Finished!!
```

Como se aprecia, en primer lugar aparecen tanto los paquetes solicitados como aquellos que se están desarrollando que aparecen en la ventana de paquetes en desarrollo de los cuales solo se instalarán las dependencias. Posteriormente ocurre un conflicto con el paquete *technology* porque se le ha pedido desinstalar, pero en la figura 47 se muestra que también es una dependencia del paquete *memory* que se pidió para instalar por lo que hay que resolver dicho conflicto. Después se efectúan las preguntas al usuario para resolver distintos conflictos: para el paquete *agent_extmem*, la pregunta es si utilizar la versión solicitada o la más alta compatible (por lo que el campo "versiones" tras obtener el árbol solo hay una); con el paquete *basic* hay versiones incompatibles; y con el *arithmetic* el conflicto se produce entre dos versiones compatibles. Tras las comprobaciones, como se ha forzado la instalación, el paquete *apb* se instalará en una versión anterior y no compatible, pero avisará al usuario (una vez por ser incompatible y otra por ser una versión inferior) y realizará la comprobación de la integridad. Después se hará lo propio con el resto de paquetes hasta formar la lista final de paquetes para instalar en la que aparece el paquete *driver_common*, que no se solicitó porque es dependencia del paquete *agent_clkrst* que se está desarrollando. El paquete *common* no aparece porque a pesar de ser una dependencia de un paquete que se desarrolla, como también se desarrolla no se debe modificar. Finalmente, tras realizar las instalaciones, se desinstala el paquete *technology* y se reescribe el fichero de instalación y actualiza el panel principal de la interfaz gráfica.

7.4. Ventana de paquetes en desarrollo

Una de las novedades que incluye la versión gráfica es la posibilidad de añadir o eliminar dependencias a los ficheros `paquete.json` que se están desarrollando sin necesidad de editar los ficheros manualmente. Para ello, existe la ventana de paquetes de desarrollo.

Cuando se inicia la aplicación, dentro del directorio `tools/myum` en el directorio de trabajo, se buscan todos los ficheros de definición de paquetes y se añaden tanto al fichero de instalación con versión `d.d.d` si no lo están y se presentan en esta ventana. Esta ventana muestra para cada paquete que se desarrolla sus dependencias. Por ejemplo, en el caso de la figura 47, el paquete *agent_clkrst* tiene como dependencias los paquetes *agent_common*, *technology*, *driver_common* y *basic* en las versiones especificadas.

Si se quisiera añadir una nueva dependencia a cualquiera de ellos para que fuera instalada al utilizar la opción de aplicar, en lugar de editar su fichero, la interfaz gráfica da la posibilidad de hacerlo haciendo clic con el botón derecho al nombre del paquete. Una vez hecho esto se abrirá una nueva ventana en la que aparecerán todos los paquetes en todas las versiones disponibles para añadir como dependencia. Al hacer doble clic en un paquete, se añadirá a la lista de dependencias siempre y cuando no esté ya en la lista aunque sea en otra versión. Tras añadir el paquete la ventana se cerrará y la dependencia añadida se verá en la ventana gráfica. Para el proceso de borrado de dependencias, simplemente se hace doble clic en el nombre y se

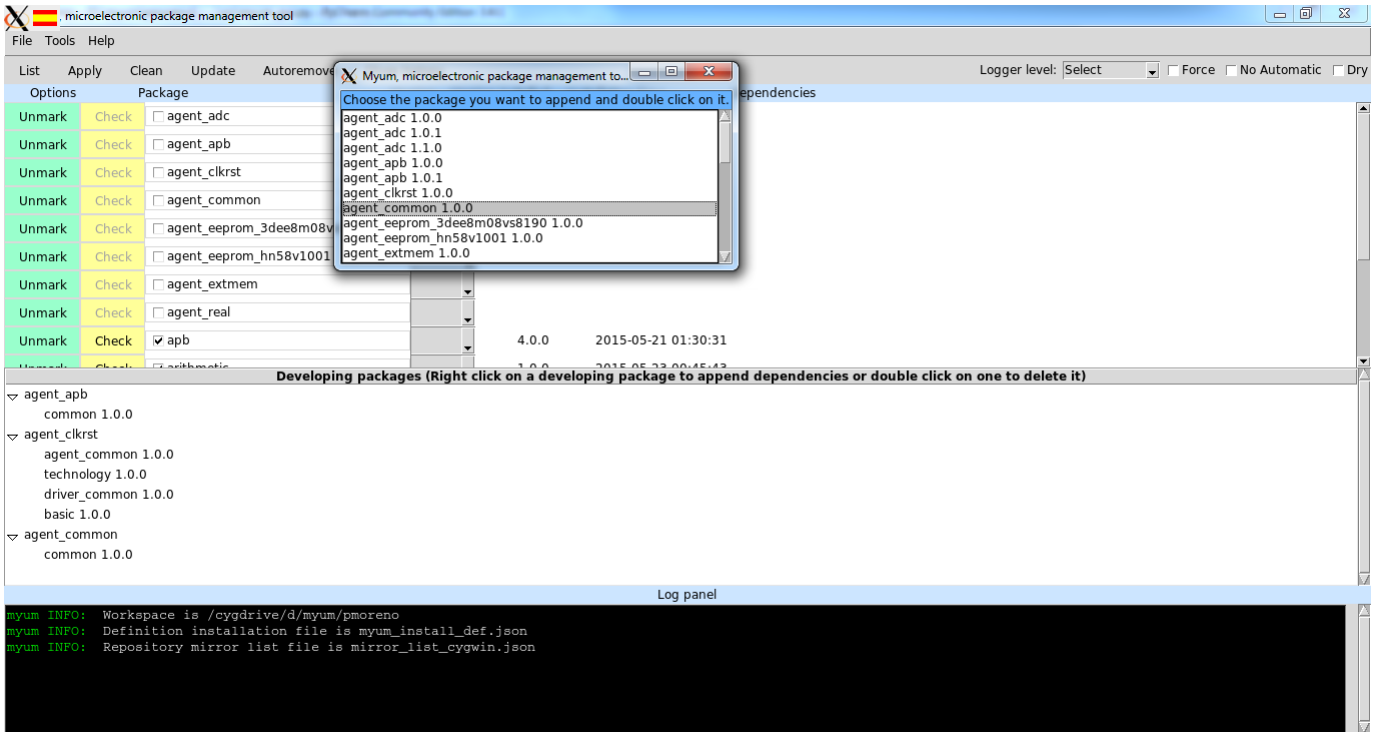

Figura 48: Ventana para añadir dependencias a un paquete mediante la interfaz gráfica.

borrará de la lista. La figura 48 muestra la ventana para añadir dependencias.

7.5. Panel de mensajes

La última de las partes de la interfaz es el panel de mensajes. El comportamiento del mismo es muy similar al de la consola utilizada en la versión por línea de comandos. Este panel, en principio no permite la introducción de texto por el usuario, salvo en los casos en los que se resuelve un conflicto en los cuales se activa la interacción para que el usuario pueda introducir la opción que prefiera. En el resto de casos, simplemente muestra los mensajes que se mostrarían en consola durante la ejecución del programa. Del mismo modo que la versión que utiliza la línea de comandos, aparte de mostrar los mensajes aquí, todos los mensajes quedan registrados en el fichero `myum.log` situado en `tools/myum` del directorio de trabajo.

8. Desarrollo complementario del proyecto y resultados

Este capítulo tiene como objeto el de presentar las tareas desarrolladas en el proyecto, no tanto para conseguir funcionalidad, que ya se han expuesto previamente, sino para conseguir completitud y que si en un futuro el proyecto debiese retomarse para añadir cualquier funcionalidad, que cualquier programador fácilmente pudiera entender y usar fácilmente el código implementado. Adicionalmente, se presentarán algunos hechos relativos a la compatibilidad del programa y se analizarán los resultados obtenidos tras finalizar el proyecto.

8.1. Pruebas del código y cobertura

A lo largo de la exposición, se han presentado las diferentes funcionalidades que presenta el programa tanto en su versión para línea de comandos como en la versión gráfica. En este capítulo, se explicarán los mecanismos utilizados para verificar el funcionamiento correcto del código y la posible detección de errores si se produjesen actualizaciones en un futuro.

Para realizar las pruebas del código se han empleado las librerías de *Python* `unittest` y `mock` [14]. La primera de ellas sirve para implementar pruebas unitarias, es decir, pruebas que sirven para comprobar que una pequeña parte del código funciona correctamente. La segunda librería sirve para reemplazar partes del sistema que se están probando por objetos que simulan el comportamiento de dichas partes. Esto es muy útil, en especial cuando se quiere probar una parte del código que requiere la interacción con el usuario. Para evitar que el programador deba interactuar cuando se realizan los bancos de pruebas, se utiliza esta librería en la cual se simulan los distintos resultados que podría haber introducido el usuario.

La filosofía llevada a cabo es realizar al menos un *test* unitario para cada una de las funciones del código intentando cubrir la mayor parte del mismo, y si con un *test* quedan casos sin cubrir, se realizaran varios con el objeto de que la cobertura sea lo mayor posible. De este modo, las funciones de mayor bajo nivel son probadas exhaustivamente y las pruebas de las funciones de nivel más alto son probadas lo suficiente para comprobar que se realizan las llamadas a las otras funciones correctamente y la integración es la correcta. Siguiendo este método, se han implementado 144 pruebas unitarias. Para su ejecución se ha diseñado un pequeño *script* que utiliza la librería `discover` para ejecutar todas las pruebas situadas en distintos ficheros dentro del directorio específico de *test*. A continuación se muestra parcialmente la salida obtenida al ejecutar el *script* para probar los *tests* unitarios.

```
test_add_pkg_to_myum_install_log (test_add_pkg_to_myum_install_log.TestAddLog)
    ... ok
test_add_pkg_to_myum_install_log_first_time (test_add_pkg_to_myum_install_log.
    TestAddLog) ... ok
test_list_all_files_failure (test_are_there_new_files.TestNewFiles) ... ok
test_list_all_files_success (test_are_there_new_files.TestNewFiles) ... ok
test_answer_no (test_ask_if_continue.TestAskContinue) ... ok
test_answer_yes (test_ask_if_continue.TestAskContinue) ... ok
                                    ...
test_uninstall_package_failure (test_uninstall_package.TestUninstallPackage) ...
     ok
```

```
test_uninstall_package_success (test_uninstall_package.TestUninstallPackage) ...
     ok
test_update_install_def (test_update_install_def.TestUpdateInstallDef) ... ok
test_update_install_def_no_file (test_update_install_def.TestUpdateInstallDef)
    ... ok
test_update_pkg_to_myum_install_log_bak (test_update_pkg_to_myum_install_log.
    TestUpdatePkgInstallLog) ... ok
test_update_pkg_to_myum_install_log_downgrade (
    test_update_pkg_to_myum_install_log.TestUpdatePkgInstallLog) ... ok
test_update_pkg_to_myum_install_log_reinstall (
    test_update_pkg_to_myum_install_log.TestUpdatePkgInstallLog) ... ok
test_update_pkg_to_myum_install_log_update (test_update_pkg_to_myum_install_log.
    TestUpdatePkgInstallLog) ... ok

----------------------------------------------------------------------
Ran 144 tests in 50.306s

OK
```

Una vez comprobados el funcionamiento apropiado de estos *tests*, para poder comprobar si son suficientes o se ha dejado gran parte del código fuera de los mismos se debe realizar una comprobación de la cobertura. La cobertura mide el porcentaje de líneas de código por las que pasa el código entre el total de líneas:

$$Cobertura = \frac{LP}{LT}$$

siendo LP el número de líneas probadas y LT el número de líneas totales

Un *test* será más eficaz cuanto mayor sea su cobertura, siendo óptimo alcanzar el 100 %. La realización de estas comprobaciones también ayuda a descubrir partes del código que pueden resultar inalcanzables (código muerto) que por ejemplo, el programador haya podido utilizar para prevenir de algún error, pero que ese problema nunca pueda llevarse a cabo porque haya alguna otra condición previa que lo evite.

Sin embargo, tampoco el resultado de la cobertura es la panacea, puesto que no indica si el programa hace lo que se quiere que haga o no: simplemente indica el ratio de líneas por las que pasan los *tests*. Si un test no está perfectamente diseñado y se supera aunque el resultado no sea el esperado (ej. puede ser que una función devuelva el mismo valor en varios casos y en la prueba se compruebe el valor únicamente y coincida, pero haya sido originado por otro motivo), podría darse una cobertura ideal sin que el funcionamiento fuera del todo correcto, aunque como herramienta de medida es útil para mejorar las pruebas unitarias.

Para obtener la cobertura, se ha utilizado la herramienta *coverage* [23], que permite medir la cobertura de programas escritos en *Python*. Este programa realiza las pruebas unitarias correspondientes y permite generar ficheros *HTML* con los resultados.

Los resultados obtenidos muestran un 98 % de cobertura en las pruebas unitarias. En este escenario, la obtención del 100 % es imposible de obtener mediante el programa, dado que existe código que únicamente se ejecuta en *Windows* y otro que solo se ejecuta en *Linux* o *Cygwin*; y del mismo modo, hay código que se ejecuta únicamente en versiones de *Python*

inferiores a la 3 y otros solo en versiones a partir de dicha versión. El programa, como mide la cobertura realizando las pruebas en un entorno concreto, solo mide la cobertura para un sistema operativo con una versión de *Python* específica, mostrando como código sin probar el resto. Para verificar el buen comportamiento, se ha comprobado los *tests* en los distintos soportes y se ha aceptado dicha cobertura porque salvaguardando las diferencias comentadas, el resultado es muy próximo al máximo.

8.2. Compatibilidad del programa

Uno de los aspectos que se comentaron al inicio es el de la compatibilidad del programa. Por una parte está la compatibilidad entre versiones de *Python* y por otra entre las distintas plataformas. En esta sección se tratarán las implicaciones que supone el hecho de que el programa deba funcionar en más de un soporte.

Respecto a la compatibilidad entre versiones de *Python*, esta es la parte menos problemática puesto que los únicos cambios producidos son el cambio de nombre de algunas librerías. En concreto, se ha cambiado el nombre de la librería que implementa las colas (utilizada para obtener la lista de paquetes), cuyo nombre pasa de `Queue` a `queue` (el único cambio es la primera letra que pasa a ser minúscula) y las librerías utilizadas para la interfaz gráfica. Este hecho simplemente se soluciona añadiendo una condición para importar las librerías necesarias según la versión, como se muestra en el ejemplo:

```
if sys.version_info[0] < 3:
    from Queue import Queue
else:
    from queue import Queue
```

Por otro lado, se encuentra la compatibilidad entre sistemas operativos, que conlleva un mayor número de cambios para que el programa se adapte correctamente. A continuación se muestran las más significativas:

1. Composición de las rutas: Mientras que *Windows* utiliza la barra invertida ('\' para su composición, en *Linux* o *Cygwin* se utiliza la barra vertical normal ('/'), por lo que se debe utilizar un mecanismo estándar que proporciona *Python* para componer rutas.

2. Repositorios locales: Cuando se define el fichero de fuentes, es posible que en lugar de definir repositorios del servidor se utilicen repositorios locales. Los mismos archivos que se encuentran de forma remota en la máquina *fs2* pueden ser accedidos desde todas las plataformas pero con nombres diferentes. Mientras que para *Windows*, el proyecto `proyecto_X` se situaría en `T:\proyecto_X`, en *Cygwin* estaría en `/cygdrive/t/proyecto_X` y en *Linux* en `/home/projects/proyecto_X`. Esto tiene el inconveniente de necesitar varios ficheros de fuentes si se quiere utilizar el mismo repositorio local desde distintas plataformas.

   Nota: Si se fija en las distintas figuras a lo largo de la exposición, se puede comprobar cómo corresponden a distintas plataformas. Una forma fácil de reconocerlas es por las rutas desde las que se ejecuta el programa, que tienen la forma descrita en el ejemplo del proyecto_X.

3. Finales de línea: Para sistemas como *Windows*, el final de línea tiene salto de línea y retorno de carro, mientras que en *Linux* solo se produce el salto de línea. Este problema puede ocasionar que las comprobaciones de integridad fallen al pasar de un sistema a otro. En realidad, este problema mediante al configuración de *Git* que ajusta estos finales no hay problema, pero si no se realizase esta configuración podría haber efectos no deseados.

4. Permisos: Los sistemas de permisos son diferentes en *Windows* y *Linux*, por lo que cuando se realice la instalación debe tenerse en cuenta que al establecer los permisos, puede ser que si la instalación se realiza en *Windows*, los permisos no queden totalmente ajustados al pasar a *Linux*. Aun así, *Python* ajusta los permisos de la forma que optimiza la compatibilidad entre sistemas.

5. Secuencias de escape: Las secuencias de escape que permiten que los mensajes aparezcan en distintos colores no están disponibles en *Windows*, por lo que para evitar que en este sistema no se muestren los códigos utilizados para dichas secuencias como texto, ya que los colores no se muestran, es necesario mantener una configuración separada al mostrar los mensajes para los distintos sistemas.

6. Comandos: Cuando se ejecutan comandos externos durante la ejecución del programa se debe tener bastante cuidado de que dichos comandos puedan ser ejecutados desde todas las plataformas ya que en caso negativo no podrán funcionar. Por ejemplo, para descomprimir se utilizan las librerías de *Python* directamente porque los comandos de descompresión de *Linux* pueden no estar en *Windows*.

8.3. Documentación

Por último, y no por ello menos importante, está la labor de documentar el código. Este paso, implícito en la propia implementación (no paso para realizar a posteriori), es de vital importancia en caso de que en un futuro otro programador quisiera realizar ampliaciones al código porque le permite entender de forma clara para qué sirve cada función o parte del código concreta.

De este paso, es de destacar que para seguir las normas de documentación de proyectos dentro de la sección debe ser posible generar ficheros *HTML* con la documentación en formato *Doxygen* [24] en lugar de únicamente en el formato que utiliza *Python*. *Doxygen* es una herramienta para generar documentación de código en multitud de lenguajes: *C*, *C++*, *Java*, *Python*, *VHDL*, *TCL*, etc. Para que todo el código de la empresa quede documentado de una forma estándar y el formato no dependa del lenguaje utilizado, se utiliza siempre esta herramienta y los comentarios deben ajustarse a ella.

8.4. Resultados

Antes de finalizar, es preciso hacer un resumen de los resultados obtenidos que engloben todo el desarrollo realizado. Al finalizar el proyecto se dispone de dos ficheros principales `myum.py` y otro `myum_gui.py`, que pueden ser ejecutados (incluso desde *Linux* se podrían

ejecutar simplemente con `./myum.py` o `./myum_gui.py`) y uno de ellos sirve para el funcionamiento mediante la línea de comandos y el otro para la interfaz gráfica.

Ambas versiones disponen de una funcionalidad común para obtener la lista de paquetes disponibles, instalar, desinstalar, actualizar paquetes, comprobar su integridad, limpiar el directorio caché y mostrar la lista de paquetes instalados y su historial, junto con las posibles modificaciones de las operaciones disponibles para seleccionar los ficheros de configuración o poner cambiar el comportamiento forzado la instalación, haciéndola no automática o utilizando el modo de simulación.

Los usuarios que han comenzado a utilizar el *myum* en sus proyectos han conseguido realizar su instalación fácilmente, aunque también han ido proponiendo sugerencias de mejora. En cuanto a dichos cambios, ya descritos, se encuentra la claridad de los mensajes de información, que en algunos casos podía resultar confusa a alguien no habituada al programa. Por otra parte, en principio se eligió el color amarillo para las advertencias. Uno de los problemas era que mientras que un desarrollador que utiliza un terminal con fondo negro veía bien el color, el que la usase con fondo blanco, podría tener problemas para ver el color. Por ello, finalmente se cambió al naranja.

Otra de las peticiones funcionales más significativas fue la de incluir la ventana de paquetes de desarrollo de la interfaz gráfica. En un inicio, solo existía el panel principal y si bien los paquetes en desarrollo se podían tener en cuenta en la instalación, no se permitía la modificación de sus dependencias de forma interactiva (era necesario la edición de los ficheros). Tras analizar la propuesta, se realizaron los cambios. Por otro lado, la implementación de estos cambios también permitió corregir errores que podían surgir si no estaba todo el entorno bien definido.

Una vez realizada la implementación de todos estos cambios se ha obtenido una herramienta robusta, que satisface las necesidades expuestas al inicio de la memoria y gracias a la cual se facilita en gran medida la tarea de configuración de los paquetes en el desarrollo de un proyecto en la empresa.

9. Conclusiones

9.1. English version

The main purpose of this Bachelor Thesis has been to design a microelectronic tool which handles *IP* packages, which allows users to perform operations as install, uninstall and so on. To achieve this goal, *myum* has been implemented in two separated but consistent versions: command-line and graphic one. Before starting to implement the code to get the functionality required, it was necessary to design a metadata structure with files to be read and written which will provide the program the required information to manage packages.

Once the metadata was defined, *Python* was used to develop the command-line version in which the first options were the options used to list the available packages and install. From those, taking into account how other package managers work according to the information obtained while preparing the state of art, new options to check integrity, uninstall, update or autoremove options were added.

After implementing the command-line version, it was possible to give the program a front-end which made it easier to use, but without removing functionality nor adding. It was important not to include anything since users who prefer command-line do not have to use graphic mode to do any operation. When all this stuff was finished, unit tests were designed to check the program and to be able to maintain it in the future.

The most challenging part of the project has been to familiarize with the projects organization in *Crisa*, *Git* and with *Python* libraries to know what it is needed to do. As regards *Git*, it is simple to know the general work as it could be similar to other revision control systems like *Subversion* but what presented more difficulties was to understand how branches and tags work in conjunction with the conventions used in the section to organize projects and how to use it to achieve the goal. As for *Python*, there is a lot of documentation of their libraries and there are lots of forums where users ask about their problems and when there was an error, it was easy to find a post where another programmer found the same issue and how to solved it. However, facts like libraries have changed between *Python 2* and *Python 3* made it more difficult to make all work and in the case of the graphic interface, it was a bit challenging at first to use its libraries because of the lack of experience. Finally, *Cygwin* was another source of issues since apart from combining *Windows* and *Linux* and the fact that it could be weird at first, configuration gave lots of problems mainly because usernames in both operative systems were different.

Myum is now being used by different workers in their projects and they consider this work has a lot of benefits when they prepare their projects. What is more, this version has a lot of new features respect the original one and some of them like restricting installation to clean repositories have had a very good acceptation. In addition, graphic user interface has made the program easier to use although experience has shown that there are clearly different types of users. Some of them always use command-line version and never open the graphic mode, whereas other just use the last one. Furthermore, the same situation happens when referring to operative systems. Computers have *Windows* installed but there is access via *ssh*

to *Linux* machines. There are users which just prefer *Windows*, while others always use *Cygwin* of *Linux*. These facts show how important was the fact to make the program available to different platforms to make every user use the program as he likes.

If objectives of this Bachelor Thesis, which were declared at the beginning, are taken into consideration, it is certain that in general they are accomplished and day-by-day experience will measure better how effective *myum* is and will tell if changes are needed in the future or not. As far as it can be said, it has been worth doing the project, not only at a personal level because of the knowledge and experience obtained, but also because this project is not something isolated for academic purposes, it is something useful that other people are going to use.

Finally, as a possible continuation of this project, (which have not been implemented because it would represent a separated project and it is not inside the scope of the Bachelor Thesis) it could be designed a tool to configure the project at the beginning which could generate and run the necessary scripts to run it. Among these scripts one could be to set the environment variables of the project as the `WORKSPACE` used in *myum*. What is more, that tool would also manage the available applications that a project can use and their specific versions in the different platforms. Between these applications, *myum* would be one of them. Therefore, both projects could link all the necessary to prepare a project for its development.

9.2. Versión en castellano

El propósito principal de este Trabajo Fin de Grado ha sido diseñar una herramienta para microelectrónica que se encargue de gestionar los paquetes de los *IPs*, que permite a los usuarios realizar operaciones como instalar, desinstalar, etc. Para lograr este objetivo, *myum* ha sido implementado en dos versiones independientes pero consistentes: una para línea de comandos y otra versión gráfica. Antes de comenzar a implementar el código para obtener la funcionalidad requerida fue necesario diseñar una estructura de metadatos con ficheros que al ser leídos y escritos proporcionasen al programa la información necesaria para poder tratar los paquetes.

Una vez que los metadatos estaban definidos, se utilizó *Python* para desarrollar la versión en la línea de comandos en la cual las primeras opciones eran aquellas para listar los paquetes e instalar. A partir de ellas y teniendo en cuenta el funcionamiento de otros gestores de paquetes con la información recopilada durante la preparación del estado del arte, se añadieron nuevas opciones para comprobar la integridad, desinstalar, actualizar o el autoborrado de paquetes.

Después de implementar la versión para línea de comandos, fue posible dotar al programa de un *front-end* que lo hacía más fácil de usar, pero sin eliminar o añadir funcionalidad. Fue muy importante no incluir nada nuevo para conseguir que los usuarios que usen siempre la consola no tengan la necesidad de usar en ningún caso la versión gráfica. Cuando todo esto se finalizó, se diseñaron las pruebas unitarias para comprobar el programa y poder mantenerlo en el futuro.

La parte más complicada del proyecto ha sido familiarizarse con la estructura de organización de proyectos en *Crisa*, *Git* y las librerías de *Python* para entender los requisitos solicitados. Respecto a *Git*, es bastante sencillo conocer su funcionamiento general, que puede ser similar al de otros sistemas de control de versiones como *Subversion*, pero lo que presentaba mayores dificultades era entender el funcionamiento de las ramas y etiquetas en conjunto con los convenios utilizados en la sección para organizar los proyectos y cómo usar todo esto para conseguir el objetivo global. Sobre *Python*, existe mucha documentación de sus librerías y hay muchos foros donde los usuarios preguntas sobre sus problemas y cuando había un error, resultaba fácil encontrar un mensaje en el que otro programador tenía el mismo problema y cómo lo resolvía. Sin embargo, algunos hechos como las librerías que cambiaban entre *Python 2* y *Python 3* complicaban todo el trabajo y en el caso de la interfaz gráfica, resultó difícil al principio usar sus librerías por la falta de experiencia con ellas. Finalmente, *Cygwin* también fue otra fuente de problemas porque aparte de combinar *Windows* y *Linux* y que pudiera resultar desconocido al principio, la configuración produjo muchos problemas principalmente porque el nombre de usuario utilizado en ambos sistemas operativos era diferente.

El programa *myum* está siendo utilizado por los distintos trabajadores en sus proyectos y lo consideran con muchos beneficios. Además, la versión final tiene un montón de características nuevas respecto a la original y algunas de ellas han tenido una gran aceptación, como la restricción de la instalación a repositorios limpios. También, la interfaz gráfica ha facilitado el uso del programa aunque la experiencia ha mostrado la existencia de diferentes tipos de usuarios. Algunos de ellos siempre utilizan la línea de comandos y nunca abren la interfaz gráfica y otros realizan justo lo contrario. Además, la misma situación ocurre cuando uno se refiere a los sistemas operativos. Los ordenadores tienen instalado *Windows*, pero pueden acceder a través de *ssh* a las máquinas de *Linux*. Hay usuarios que simplemente prefieren usar *Windows*, mientras que otros siempre intentan utilizar *Cygwin* o *Linux*. Estos hechos muestran cómo de importante era el hecho de diseñar el programa para que fuera compatible con las distintas plataformas para hacer que cada usuario use el programa de la forma que prefiera.

Si los objetivos de este Trabajo Fin de Grado que fueron planteados al inicio se tienen en cuenta, ciertamente se han cumplido los objetivos generales y la experiencia cotidiana será la que mide la efectividad final que tenga el *myum* y dirá si se requieren cambios en un futuro o no. Por lo que se puede decir hasta el momento, ha merecido la pena realizar el proyecto, no solo a nivel personal por la experiencia y conocimiento obtenido, sino porque este proyecto no es un proyecto aislado para propósitos académicos, sino es algo útil que otras personas van a utilizar.

Finalmente, como posible continuación al proyecto (que no ha sido implementado porque representaría un trabajo totalmente separado y no entra dentro del alcance de este Trabajo Fin de Grado), se podría diseñar una herramienta para configurar un proyecto al inicio que pudiera generar y ejecutar los *scripts* necesarios para ponerlo en marcha. Entre estos *scripts*, uno podría ser para definir las variables de entorno, como la `WORKSPACE` que utiliza el *myum*. Además, esta herramienta también controlaría las aplicaciones que el proyecto puede usar y sus versiones específicas en las distintas plataformas. Entre esas aplicaciones, el *myum* sería una de ellas. Por tanto, ambos proyectos podrían unir todo lo necesario para preparar un proyecto para su desarrollo.

Anexos

A. Planificación del proyecto

La realización del Trabajo Fin de Grado ha requerido sucesión de una serie de tareas a lo largo de los meses en los que se ha llevado a cabo su ejecución. En esta sección se presentan las distintas tareas, junto con sus relaciones y precedencias.

A.1. Estructura del desglose del trabajo (*EDT*)

Para empezar, se presenta la Estructura del desglose del trabajo (*EDT*, o también conocida por su nombre en inglés *Work Breakdown Structure* (*WBS*)) [25], que es una descomposición jerárquica en la cual se dividen los entregables y el trabajo del proyecto en partes más pequeñas, de modo que estas sean más fáciles de manejar. En este proyecto se diferencian seis hitos principales de los cuales cada uno tiene diversas tareas. La figura 49 muestra este esquema.

A.2. Secuenciación de las tareas

Sobre las distintos hitos del proyecto y sus respectivas tareas se establece una estimación de las duraciones y de la forma en la que deben ir secuenciadas para tener en cuenta las relaciones de precedencia entre las distintas tareas. A continuación se muestra la descripción de los hitos y tareas:

- Hito 1: Preparación técnica al proyecto: El objetivo que debe alcanzarse en este hito es que el alumno conozca las herramientas que debe utilizar en el proyecto y comprenda las necesidades del mismo.
  - Tarea 1.1: Aprendizaje de *Python*: Durante esta tarea, el alumno aprende a utilizar la sintaxis propia del lenguaje e implementa ejemplos de dificultad ascendente para poder enfrentarse al proyecto real.
    Duración: 1 semana.
    Precedencias: Se puede realizar al inicio del proyecto.
  - Tarea 1.2: Entendimiento del problema planteado: En esta fase se parte del código de la versión original del programa en *TCL* y se traduce a *Python* para que así se profundice en el lenguaje y se analicen todos los algoritmos para comprender el programa.
    Duración: 4 semanas.
    Precedencias: Tarea 1.1.
  - Tarea 1.3: Aprendizaje del sistema de control de versiones *Git*: Al igual que en la tarea 1.1, puesto que es necesario conocer este sistema para implementar el programa,

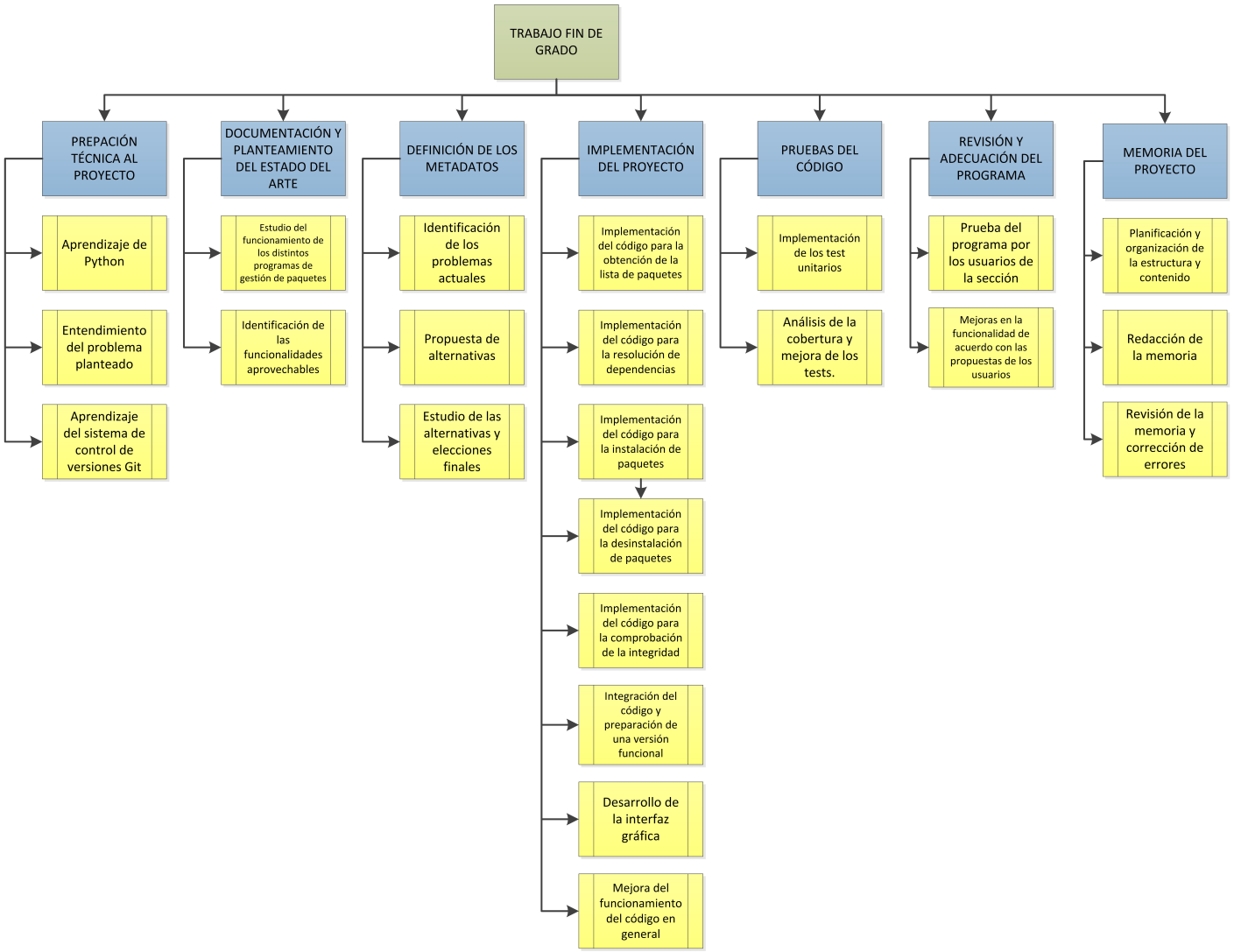

Figura 49: Estructura de descomposición del trabajo.

se necesita documentarse al respecto.
Duración: 1 semana.
Precedencias: Se puede realizar al inicio del proyecto.

- Hito 2: Documentación y planteamiento del estado del arte: El objetivo es conocer las distintas alternativas existentes en los sistemas de gestión de paquetes para poder dotar de mayor funcionalidad al programa existente y entender mejor sus problemas para mejorarlos teniendo en cuenta las soluciones actuales.

  - Tarea 2.1: Estudio del funcionamiento de los distintos programas de gestión de paquetes: Esta tarea consiste en la documentación sobre los distintos sistemas para conocer el funcionamiento general al mayor detalle posible.
    Duración: 1.5 semanas.
    Precedencias: Tarea 1.2
  - Tarea 2.2: Identificación de las funcionalidades aprovechables: Esta tarea trata de poner en conjunto el conocimiento previo del proyecto con lo que se ha aprendi-

do sobre los sistemas de gestión de paquetes para identificar lo que se puede utilizar en el proyecto.
Duración: 0.5 semanas.
Precedencias: Tarea 2.1.

- Hito 3: Definición de los metadatos: Al llegar a este hito, se debe conocer el formato definitivo de los ficheros de metadatos que tiene el proyecto.
  - Tarea 3.1: Identificación de los problemas actuales: La tarea trata de comprender qué elementos de los metadatos actuales son problemáticos para poder mejorarlos.
    Duración: 0.5 semanas.
    Precedencias: Tarea 1.2.
  - Tarea 3.2: Propuesta de alternativas: Con los problemas identificados en los metadatos, se diseñan diferentes posibles alternativas que supongan una solución a las dificultades encontradas.
    Duración: 0.5 semanas.
    Precedencias: Tareas 2.2 y 3.1.
  - Tarea 3.3: Estudio de las alternativas y elecciones finales: Para esta tarea se parte del código implementado de pruebas en la fase inicial y se analiza qué ventajas e inconvenientes tendrían las distintas alternativas para tomar la decisión final para cada fichero.
    Duración: 1 semana.
    Precedencias: Tarea 3.2.

- Hito 4: Implementación del proyecto: Al alcanzar este hito se debe disponer de una versión del programa con todas las funcionalidades.
  - Tarea 4.1: Implementación del código para la obtención de la lista de paquetes: Esta tarea trata de diseñar el código necesario para poder conocer todos los paquetes disponibles en los distintos repositorios y acceder a los datos de *Git*.
    Duración: 1 semana.
    Precedencias: Tareas 1.3 y 3.3.
  - Tarea 4.2: Implementación del código para la resolución de dependencias: Esta tarea consiste en diseñar la forma óptima e implementarla para la resolución de las dependencias.
    Duración: 3 semanas.
    Precedencias: Tarea 3.3.
  - Tarea 4.3: Implementación del código para la instalación de paquetes: Esta tarea consiste en implementar el código necesario para poder instalar un paquete disponible de la forma más sencilla (sin comprobar la integridad, ni desinstalando contenido, sino que sobrescribiéndolo)
    Duración: 1.5 semanas.
    Precedencias: Tareas 4.1, 4.2,

- Tarea 4.4: Implementación del código para la desinstalación de paquetes: Esta tarea incluye el proceso de desinstalación, así como la integración de este proceso en algunos casos dentro de la instalación.
  Duración: 0.5 semanas.
  Precedencias: Tarea 4.3.
- Tarea 4.5: Implementación del código para la comprobación de la integridad: La tarea comprende el proceso para implementar el código que comprueba si un paquete ha sido modificado y la integración de esta funcionalidad en los procesos de instalación y desinstalación.
  Duración: 0.5 semanas.
  Precedencias: Tarea 4.4.
- Tarea 4.6: Integración del código y preparación de una versión funcional: La tarea consiste en juntar todo el código implementado anteriormente para tener una primera versión que funcione mediante la línea de comandos.
  Duración: 1 semana.
  Precedencias: Tarea 4.5.
- Tarea 4.7: Desarrollo de la interfaz gráfica: La tarea consiste en implementar el código necesario para que toda la funcionalidad existente mediante la línea de comandos lo sea también de forma gráfica. Esta tarea, por mantenimiento de las versiones no se puede realizar hasta que no existe una primera versión con toda la funcionalidad para la línea de comandos.
  Duración: 3 semanas.
  Precedencias: Tarea 4.6.
- Tarea 4.8: Mejora del funcionamiento del código en general: Esta tarea tiene el objetivo de encontrar casos fuera de lo común donde el programa podría fallar e implementar las excepciones necesarias para evitar los problemas, así como introducir alguna nueva funcionalidad complementaria que mejore el *myum*.
  Duración: 1 semana.
  Precedencias: Tarea 4.6.

- Hito 5: Pruebas del código: Al alcanzar este hito se deben disponer los *tests* unitarios que verifiquen el funcionamiento del programa.

  - Tarea 5.1: Implementación de los *tests* unitarios: La tarea consiste en implementar el código necesario para poder probar el programa.
    Duración: 2 semanas.
    Precedencias: Tareas 4.7 y 4.8.
  - Tarea 5.2: Análisis de la cobertura y mejora de los *tests*: La tarea consiste en medir las líneas de código por las que pasan o no pasan las pruebas y aumentar la cobertura con nuevos *tests*.
    Duración: 1 semana.
    Precedencias: Tareas 5.1.

- Hito 6: Revisión y adecuación del programa: Al alcanzar este hito, el programa debe estar totalmente terminado y en funcionamiento.

- Tarea 6.1: Prueba del programa por los usuarios de la sección: La tarea consiste en que con el código ya implementado y funcional, la sección lo comience a usar y que reporte de los posibles problemas o aspectos mejorables.
  Duración: 5 semanas.
  Precedencias: Tareas 4.7 y 4.8.
- Tarea 6.2: Mejoras en la funcionalidad de acuerdo a las propuestas de los usuarios: La tarea consiste en realizar las modificaciones tras la revisión del programa.
  Duración: 5 semanas.
  Precedencias: Tareas 4.7 y 4.8.

- Hito 7: Memoria del Proyecto: Al alcanzar este hito se debe tener la terminada la memoria para poder presentarse a la defensa del Trabajo Fin de Grado.
  - Tarea 7.1: Planificación y organización de la estructura y contenido: Consiste en pensar qué partes va a tener la memoria y qué contenido debe ir en cada una de ellas.
    Duración: 0.5 semanas.
    Precedencias: Tareas 4.7 y 4.8.
  - Tarea 7.2: Redacción de la memoria: Consiste en elaborar el documento que se va a presentar como Trabajo Fin de Grado.
    Duración: 4 semanas.
    Precedencias: Tarea 7.1.
  - Tarea 7.3: Revisión de la memoria y corrección de errores: Esta tarea final consiste en revisar la memoria y a partir del *feedback* del tutor, realizar las modificaciones pertinentes para la presentación final del proyecto.
    Duración: 3 semanas.
    Precedencias: Tareas 5.2, 6.2 y 7.2.

Para resumir toda la información presentada se muestra la siguiente tabla 6. En ella aparecen todas las tareas junto con sus duraciones y precedencias. Para mayor claridad, también se han incluido los hitos, aunque sin información de duración ya que estos no tienen duración. La duración total del proyecto obtenida, asimismo es de 26 semanas, lo que equivale a unos 6.5 meses.

A.3. Diagrama de Gantt

Con la información descrita de las tareas se puede elaborar el Diagrama de Gantt [26], que se muestra en la figura 50. En él se contemplan las tareas críticas (aquellas cuyo retraso implica un retraso en el tiempo total de ejecución del proyecto) en rojo. Los recursos para el proyecto son de básicamente un estudiante de último año del Grado en Ingeniería en Tecnologías de Telecomunicación, si bien en la tarea 6.1 colaboran otros trabajadores y durante todo el proyecto están presentes el tutor académico (principalmente en las partes 2 y 7) y el tutor de la empresa (principalmente en las partes 1-6). Este hecho, junto con las restricciones del proyecto conducen a que no se pueda realizar demasiada paralelización de tareas (sobre todo al inicio) y que aparezcan muchas tareas en el camino crítico.

Diccionario EDT	Descripción	Duración (semanas)	Precedencias
1	**Preparación técnica al proyecto**		
1.1	Aprendizaje de Python	1	-
1.2	Entendimiento del problema planteado	4	1.1
1.3	Aprendizaje del sistema de control de versiones Git	1	-
2	**Documentación y planteamiento del estado del arte**		
2.1	Estudio del funcionamiento de los distintos programas de gestión de paquetes	1.5	1.2
2.2	Identificación de las funcionalidades aprovechables	0.5	2.1
3	**Definición de los metadatos**		
3.1	Identificación de los problemas actuales	0.5	1.2
3.2	Propuesta de alternativas	0.5	2.2 y 3.1
3.3	Estudio de las alternativas y elecciones finales	1	3.2
4	**Implementación del proyecto**		
4.1	Implementación del código para la obtención de la lista de paquetes	1	1.3 y 3.3
4.2	Implementación del código para la resolución de dependencias	3	3.3
4.3	Implementación del código para la instalación de paquetes	1.5	4.1 y 4.2
4.4	Implementación del código para la desinstalación de paquetes	0.5	4.3
4.5	Implementación del código para la comprobación de la integridad	0.5	4.4
4.6	Integración del código y preparación de una versión funcional	1	4.5
4.7	Desarrollo de la interfaz gráfica	3	4.6
4.8	Mejora del funcionamiento del código en general	1	4.6
5	**Pruebas del código**		
5.1	Implementación de los tests unitarios	2	4.7 y 4.8
5.2	Análisis de la cobertura y mejora de los tests	1	5.1
6	**Revisión y adecuación del programa**		
6.1	Prueba del programa por los usuarios de la sección	5	4.7 y 4.8
6.2	Mejoras en la funcionalidad de acuerdo a las propuestas de los usuarios	5	4.7 y 4.8
7	**Memoria del proyecto**		
7.1	Planificación y organización de la estructura y contenido	0.5	4.7 y 4.8
7.2	Redacción de la memoria	4	7.1
7.3	Revisión de la memoria y corrección de errores	3	5.2, 6.2 y 7.2
	DURACIÓN TOTAL DEL PROYECTO	26	

Tabla 6: Resumen de tareas, duraciones y precedencias del proyecto.

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**TRABAJO FIN DE GRADO**	**130 días**	**lun 22/12/14**	**vie 19/06/15**
2	**1. Preparación técnica al proyecto**	**25 días**	**lun 22/12/14**	**vie 23/01/15**
3	1.1. Aprendizaje de Python	1 sem	lun 22/12/14	vie 26/12/14
4	1.2. Entendimiento del problema planteado	4 sem.	lun 29/12/14	vie 23/01/15
5	1.3. Aprendizaje del sistema de control de versiones Git	1 sem	lun 22/12/14	vie 26/12/14
6	**2. Documentación y planteamiento del estado del arte**	**10 días**	**lun 26/01/15**	**vie 06/02/15**
7	2.1. Estudio del funcionamiento de los distintos programas de gestión de paquetes	1,5 sem.	lun 26/01/15	mié 04/02/15
8	2.2. Identificación de las funcionalidades aprovechables	0,5 sem.	mié 04/02/15	vie 06/02/15
9	**3. Definición de los metadatos**	**17,5 días**	**lun 26/01/15**	**mié 18/02/15**
10	3.1. Identificación de los problemas actuales	0,5 sem.	lun 26/01/15	mié 28/01/15
11	3.2. Propuesta de alternativas	0,5 sem.	lun 09/02/15	mié 11/02/15
12	3.3. Estudio de las alternativas y elecciones finales	1 sem	mié 11/02/15	mié 18/02/15
13	**4. Implementación del proyecto**	**47,5 días**	**mié 18/02/15**	**vie 24/04/15**
14	4.1. Implementación del código para la obtención de la lista de paquetes	1 sem	mié 18/02/15	mié 25/02/15
15	4.2. Implementación del código para la resolución de dependencias	3 sem.	mié 18/02/15	mié 11/03/15
16	4.3. Implementacióndel código para la instalación de paquetes	1,5 sem.	mié 11/03/15	vie 20/03/15
17	4.4. Implementación del código para la desinstalación de paquetes	0,5 sem.	lun 23/03/15	mié 25/03/15
18	4.5. Implementación del código para la comprobación de la integridad	0,5 sem.	mié 25/03/15	vie 27/03/15
19	4.6.Integración del código y preparación de una versión funcional	1 sem	lun 30/03/15	vie 03/04/15
20	4.7. Desarrollo de la interfaz gráfica	3 sem.	lun 06/04/15	vie 24/04/15
21	4.8. Mejora del funcionamiento del código en general	1 sem	lun 06/04/15	vie 10/04/15
22	**5. Pruebas del código**	**15 días**	**lun 27/04/15**	**vie 15/05/15**
23	5.1. Implementación de los tests unitarios	2 sem.	lun 27/04/15	vie 08/05/15
24	5.2. Análisis de la cobertura y mejora de los tests	1 sem	lun 11/05/15	vie 15/05/15
25	**6. Revisión y adecuación del programa**	**25 días**	**lun 27/04/15**	**vie 29/05/15**
26	6.1. Prueba del programa por los usuarios de la sección	5 sem.	lun 27/04/15	vie 29/05/15
27	6.2. Mejoras en la funcionalidad de acuerdo a las propuestas de los usuarios	5 sem.	lun 27/04/15	vie 29/05/15
28	**7. Memoria del proyecto**	**40 días**	**lun 27/04/15**	**vie 19/06/15**
29	7.1. Planificación y organización de la estructura y contenido	0,5 sem.	lun 27/04/15	mié 29/04/15
30	7.2. Redacción de la memoria	4 sem.	mié 29/04/15	mié 27/05/15
31	7.3. Revisión de la memoria y corrección de errores	3 sem.	lun 01/06/15	vie 19/06/15

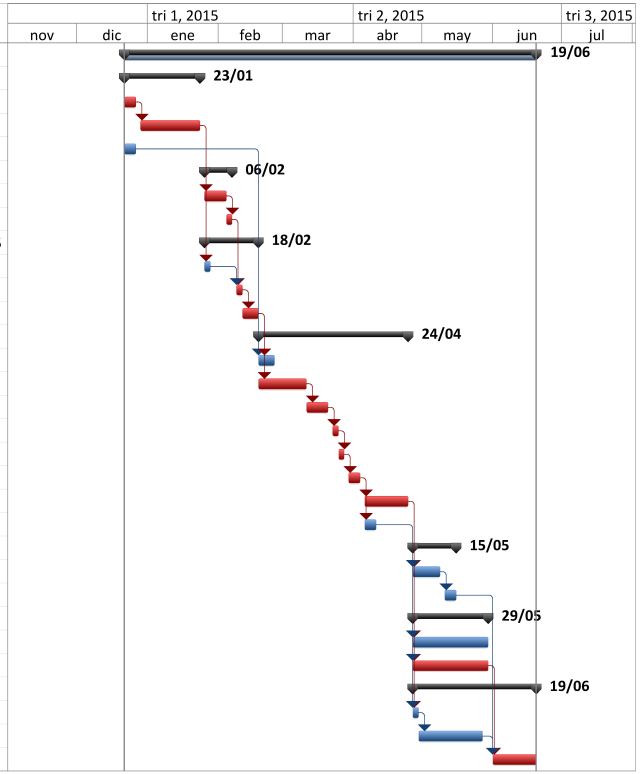

Figura 50: Diagrama de Gantt del proyecto.

B. Presupuesto del proyecto

En esta sección se presenta un presupuesto detallado de los distintos recursos utilizados para la elaboración del proyecto. Se distinguen entre costes directos, entre los que se encuentran los recursos materiales y humanos; y los costes indirectos.

B.1. Recursos materiales

La realización del proyecto básicamente se puede realizar con un ordenador. En concreto se han utilizado dos ordenadores, uno de sobremesa utilizado en la empresa y otro portátil para su uso fuera de ella. Las características de estos equipos son:

- Ordenador de sobremesa: Procesador *Intel ® Pentium ® Dual CPU E2180* 2.0 GHz. 4GB *RAM*. Disco duro de 75GB. Sistema operativo *Windows 7 Professional*.
- Ordenador portátil: Procesador *Intel ® Core™i7-3612QM* 2.10GHz. 6GB RAM. Disco duro de 750GB. Tarjeta gráfica *AMD Radeon HD 7670M*. Sistema operativo *Windows 7 Home Edition*.

Por otro lado, se encuentra el *software* utilizado en los equipos comentados. En la siguiente lista se encuentra todo el *software* utilizados:

- Sistema operativo *Windows* (y sus programas incorporados): Requiere licencia.
- Sistema operativo Linux (y sus programas incorporados): *Software* libre.
- *Cygwin*: *Software* libre.
- *Python*: *Software* libre.
- *Pycharm* [27] (Entorno de desarrollo integrado): *Software* libre.
- Git: *Software* libre.
- SourceTree (cliente gráfico de Git): *Software* libre.
- *Coverage*: *Software* libre.
- *Doxygen*: *Software* libre.
- *Texmaker*: *Software* libre.
- *MiKTeX*: *Software* libre.
- *Microsoft Office 2010* (*suite* completa): Requiere licencia
- *Adobe Reader*: *Software* libre.

De toda la lista de *software*, dado que la mayoría es libre, solo es necesario amortizar el sistema operativo *Windows* y la *suite* ofimática. En el caso del ordenador portátil, al estar incluidos ambos en el precio del ordenador, se realizará la amortización conjunta con el *hardware*. En el ordenador de sobremesa, *Microsoft Office* no se ha utilizado, por lo que solo hay que amortizar la licencia de *Windows*, aunque dado que la empresa adquiere las licencias a gran escala y el precio es relativamente bajo (y su amortización en el periodo del proyecto mucho menor), se considerará este coste como indirecto.

Respecto al *hardware*, se calculará la amortización de forma lineal. El periodo de amortización se estima en unos 4 años y el precio de los ordenadores de sobremesa y portátil asciende a unos 500€y 700€, respectivamente. La amortización se calcula entonces como:

$$Amortización = \frac{Precio \ \cdot \ Duración \ del \ proyecto}{Tiempo \ de \ amortización}$$

Con estos datos, se obtiene unos valores de amortización en el periodo de 6,5 meses que dura el proyecto de 67,71€y 94,79€para el ordenador de sobremesa y portátil, respectivamente.

B.2. Recursos humanos

En relación a los recursos humanos, el proyecto lo ha elaborado un estudiante de cuarto curso en el Grado en Ingeniería en Tecnologías de Telecomunicación, cuyo coste se reparte entre las horas presenciales en la empresa y las horas de trabajo en casa. Aparte del trabajo del alumno, se encuentra el realizado por el director y el tutor del proyecto. La labor de pruebas del programa de otros trabajadores no se computa al presupuesto del proyecto, ya que no es una tarea explícita a este proyecto, sino una tarea computable a otro proyecto que se aprovecha al mismo.

Respecto del trabajo del director del proyecto, se estima que su dedicación ha sido de unas 90 horas y que el coste por hora de su trabajo es de 75€/h, lo que da un total de 6.750,00€. En relación al tutor, su dedicación ha sido de en torno a 45 horas, con un coste de 70 euros la hora, que proporciona un total de 3.150,00€.

Para calcular el importe del trabajo del alumno, se establecerá un coste por hora de trabajo de 35€/h y se calculará el importe a partir del desempeño realizado. Se estima un total de unas 325h de trabajo presenciales en la empresa y otras 100h de trabajo fuera de la misma, por lo que el total de horas asciende a 425h y el importe es de 14.875,00€.

B.3. Presentación del presupuesto

Para terminar de detallar los costes faltarían los costes indirectos. Entre estos podemos encontrar la conexión a Internet, la electricidad, agua, licencias de la empresa, etc. Para el cálculo de estos costes, se estima un importe del 20 % de la suma de todos los costes directos. De este modo, se presenta el presupuesto finalmente en la tabla 7, donde el total del proyecto asciende a 29.925,00€.

Concepto	Cantidad	Coste unitario	Tiempo	Vida útil	Total
Recursos materiales					
Ordenador de sobremesa	1 ud.	500€	6,5 meses	4 años	67.71€
Ordenador portátil	1 ud.	700€	6,5 meses	4 años	94.79€
Recursos humanos					
Trabajo estudiante del Grado en Ingeniería en Tecnologías de Telecomunicación	425h	35€/h	6,5 meses	-	14.875,00€
Trabajo del director del proyecto	90h	75€/h	6,5 meses	-	6.750,00€
Trabajo del tutor del proyecto	45h	70€/h	6,5 meses	-	3.150,00€
COSTES DIRECTOS TOTALES					24.937,50€
Costes indirectos (20 %)					4.987,50€
TOTAL					29.925,00€

Tabla 7: Presupuesto del proyecto.

C. Marco regulador

La realización de un proyecto ingenieril no implica únicamente el desarrollo técnico del mismo, sino que la solución propuesta debe adaptarse a las regulaciones técnicas y legales que se establezcan tanto a nivel interno como a nivel del sector en cualquier ámbito. Esta sección trata de explicar cuáles son las restricciones a las cuales se ha debido ajustar el proyecto.

C.1. Marco legal

En primer lugar se tratarán las medidas regulatorias a nivel legal. Dado que el proyecto se realiza para la empresa y es de carácter interno, no aplican muchas leyes que puedan restringir la capacidad del desarrollo del *software*. No obstante, como toda obra está sujeta a la Ley de Propiedad Intelectual (*LPI*). La propiedad intelectual es el conjunto de derechos que pertenecen a los autores y otros titulares (artistas, productores...) sobre las obras y prestaciones que sean fruto de su creación. [28]. En este caso, aunque el alumno sea el autor, se produce una transmisión legal de los derechos de explotación a la empresa de acuerdo al artículo 97.4 *LPI*, que indica la cesión en los casos de que el autor sea un trabajador asalariado de un programa de ordenador.

También, otra de las normativas referentes al sector *TIC* que se debe cumplir es la Ley Orgánica de Protección de Datos (*LOPD*), cuyo objeto es garantizar y proteger, en lo que concierne al tratamiento de los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente de su honor e intimidad personal y familiar [29]. En la actualidad, se debe tener bastante cuidado, sobre todo con las aplicaciones de *Big Data* o en las redes sociales, para garantizar el cumplimiento de esta normativa [30]. En nuestra aplicación particular, como no se registran datos personales, se garantiza la privacidad de todos los que utilizan el programa y por lo tanto se cumple la normativa vigente.

C.2. Marco técnico

En referencia al apartado técnico, a nivel externo no existen medidas que supongan una restricción para la solución del proyecto, como lo podría ser las bandas de frecuencia para diseños de comunicaciones. Al utilizarse ese programa únicamente en la empresa y no interferir con elementos externos, las únicas regulaciones técnicas vienen dadas por las normas de realización de proyectos internas.

Entre estas normas, para empezar se encuentra la localización del proyecto. Todos los proyectos deben llevar control de versiones mediante *Git* y se utiliza la máquina *VMGitLab* para almacenar el contenido. Este proyecto, también debe seguir el mismo procedimiento y debe utilizar el formato de *tags* establecido para que todo el contenido sea consistente.

Por otra parte, respecto a la documentación, para mantener consistencia entre todo el código de los distintos proyectos debe generarse mediante *Doxygen* y mediante la forma estándar que proporcione el lenguaje (en este caso *Python*, pero en otro proyecto pudiera ser

cualquier otro).

Otra de las restricciones técnicas, aunque no tanto a nivel regulatorio, ha sido la del *software* disponible. El programa debía ser capaz de funcionar en las distintas plataformas utilizadas (*Windows*, *Linux* y *Cygwin*) y además debía tener en cuenta que en cada una de ellas *Python* podría estar instalado en una versión diferente y no se podía forzar a tener unas condiciones particulares para el funcionamiento. El resto de condiciones técnicas que puedan haber surgido durante el proyecto al ser específicas a una parte del diseño concreto se encuentran detalladas a lo largo de toda la exposición para que puedan estar situadas dentro de su contexto.

Por otra parte, la realización del proyecto supondrá una nueva regulación técnica para el desarrollo de otros proyectos de *crips*, dado que los desarrolladores se verán obligados a elaborar, con los formatos definidos en este proyecto, los metadatos necesarios para que su proyecto esté disponible y sea instalable con el *myum*.

D. Entorno socio-económico

Uno de los aspectos de notable importancia en el desarrollo de proyectos es el entorno en el que se ubica, puesto que este afectará directa o indirectamente al desarrollo del proyecto. En esta sección se tratan los aspectos relativos al entorno socio-económico y los beneficios que supone el trabajo realizado.

D.1. Entorno social

Para entender este proyecto desde un punto de vista social, lo primero será enfocarlo hacia el segmento al que se dirige. En vista de que el proyecto es de carácter interno y sus resultados no sean directamente aplicables al exterior (si bien el concepto e ideas podrían ser útiles para otro desarrollador que quisiese realizar algo similar), no se puede identificar un segmento de la población total, sino más bien debe centrarse en aquellos potenciales usuarios que son única y exclusivamente los trabajadores de Microelectrónica de *Crisa*.

En relación a ellos, el proyecto sí que supone grandes beneficios porque simplifica y automatiza todo el proceso necesario para obtener los paquetes necesarios para el desarrollo de un proyecto. El trabajo realizado permite evitar tiempo y sobre todo problemas a sus usuarios y además da la posibilidad a sus usuarios de poder focalizarse más en el núcleo de su proyecto en vez de en configuraciones previas, lo que aumenta el rendimiento general. Si bien, este proyecto no tiene demasiado impacto social, el desarrollo de otros sí que lo tiene y propiciar su mejor desarrollo, en definitiva, es mejorar la situación para el crecimiento y avance de la tecnología en la empresa.

D.2. Entorno económico

Del mismo modo que se ha tratado el entorno social, en aspectos de economía, solo se debe focalizar el proyecto en el entorno de la empresa, ya que al ser de uso interno, no podría ser comercializado externamente para obtener beneficios económicos. Por ello, para entender los beneficios económicos, lo mejor es conocer el modelo de negocio de la empresa para así poder encuadrar el proyecto dentro del mismo.

D.2.1. Descripción de la empresa y modelo de negocio

La empresa Computadoras, Redes e Ingeniería, S.A. (*Crisa*) es una empresa española fundada en el año 1985 que se dedica al diseño y fabricación de equipos electrónicos y del software necesario para las aplicaciones del espacio: lanzadores, satélites, infraestructura orbital y vehículos de transporte espacial. También se realizan proyectos para estaciones de tierra.

En el año 2000, con la creación del consorcio de *EADS* (*European Aeronautic Defense and Space*), *Crisa* quedó como una subsidiaria con el 100 % de la propiedad de la división espa-

ñola de EADS, Astrium. Tras la reestructuración llevada a cabo a finales de 2013, actualmente la compañía es una subsidiaria de la nueva división del grupo *Airbus*, llamada *Airbus Defence and Space*, que agrupa las antiguas *Cassidian*, *Airbus Military* y *Astrium*.

Actualmente, la compañía posee una gran reputación por su tecnología en aviónica para satélites, lanzadores y vehículos espaciales y por los segmentos en tierra para aplicaciones tanto civiles como de defensa. Es de destacar la contribución que ha realizado con más de 750 unidades de vuelo en programas de la Agencia Espacial Europea (*ESA*).

Todos los equipos electrónicos diseñados en la empresa tienen un papel importante en el monitorizado de las condiciones medioambientales de la Tierra, la investigación científica del universo y la observación de eventos ambientales y catastróficos en el planeta. De entre todos los equipos, son de resaltar los computadores de abordo, los procesadores de datos, los sistemas de alimentación y distribución de potencia o las unidades de control y accionamiento, puesto que el desarrollo de los mismos ha contribuido en programas como los Meteosat de Segunda Generación, *Envisat*, *Mars* y *Venus Express*, *Planck*, *Rosetta*, *Herschel*, *GOCE*, los vehículos *ATV*, el rover *Curiosity* de la *NASA*, *Gaia* o los lanzadores Ariane 5 y Vega. De entre estos proyectos citados, ha sido de gran repercusión en los últimos meses el proyecto *Rosetta*, debido al aterrizaje del módulo *Philae* de la sonda espacial en la superficie de un cometa, y en el que *Crisa* ha desarrollado sistemas fundamentales para el posicionamiento de la sonda.

Entre los proyectos actuales de la compañía se encuentran los relacionados con las futuras misiones de Ingenio, Paz, *LISA Pathfinder*, *Sentinel*, *BepiColombo*, *EarthCARE*, *Solar Orbiter* y los nuevos satélites *Meteosat* de Tercera Generación; y los relacionados con las plataformas de satélites de comunicaciones de *Eurostar 3000*, *AlphaBus* y *SmallGEO*. [31]

El modelo de negocio de *Crisa* se basa en la venta de equipos electrónicos para uso espacial. La empresa lleva a cabo todo el proceso: desde la fase de diseño y fabricación, a las pruebas de los equipos antes de la entrega el cliente. Por ello, existen tres departamentos funcionales dedicados al diseño, producción y verificación de los productos. Entre las líneas de negocio, por una parte existe una importante inversión de dinero en *I+D*(Investigación y desarrollo) de productos innovadores, pero también se fabrica de forma recurrente equipos para determinados proyectos, como la electrónica secuencial para el lanzador europeo *Ariane 5*.

Entre los compradores que tiene la empresa, el principal es la Agencia Espacial Europea (*ESA*), que genera el 60 % de los beneficios. Sin embargo, la dependencia con el presupuesto que el Gobierno de España realiza con la *ESA* está propiciando una caída general en el sector que afecta de forma significativa a la empresa, pues cuando la Agencia Espacial Europea asigna los contratos a las distintas entidades, lo hace en función de la contribución de los distintos estados y la recesión española, junto con la falta de inversión en el sector espacial, está provocando que *Crisa* reciba menos contratos para la realización de proyectos. Aun así, la facturación en el último año es de unos 44 millones de euros y la empresa sigue siendo uno de los principales proveedores de productos electrónicos para el ámbito espacial a nivel europeo.

D.3. Aportación al modelo de negocio

Dentro del complejo modelo de negocio de la empresa y en particular, de los procesos llevados a cabo, el proyecto se enmarca dentro del departamento de ingeniería, en la sección de microelectrónica, ya que se trata de una herramienta soporte que ayuda durante el desarrollo de la electrónica de los equipos que la empresa fabrica dentro de su contexto económico.

Es evidente que la importancia de la herramienta no es comparable con la de los propios equipos que son los que realmente dan el funcionamiento en las misiones con las que trabaja la empresa, si bien cualquier aportación desde el nivel que sea es importante para la consecución de los objetivos. Incluso hasta el *software* libre que la empresa utiliza aporta a la mejora del desarrollo de los procesos y en concreto esta aplicación, que está desarrollada a medida con las características particulares de la organización de la sección, satisface las necesidades y contribuye a la mejora de la eficiencia para el desarrollo de los proyectos, lo que tiene consecuencias finales en el objetivo final de la empresa, que es aumentar su valor.

E. Summary in English

E.1. Introduction

Engineering companies usually develop lots of projects which could be very different but some of them could have elements in common as well as the structure. Particularly, Microelectronic section of *Crisa* (Computadoras, Redes e Ingeniería, S.A) works in the development of *FPGAs* (Field Programmable Gate Array) and *ASICs* (Application-Specific Integrated Circuit) designs. While developing them, there are logic blocks, called *IP* (Intelectual Property) which can be reused in different projects.

Package managers allow to easily installing software components so that a list of available packages is displayed and user chooses what packages to install, so they can be very useful to install different *IPs*. Therefore, the motivation of the Bachelor Thesis is to implement a package manager system, which allows different users to install packages corresponding to different *IPs* to be able to use them in the developing projects which requires them, so that if the package required also needs other packages, those other packages, called dependencies, are also installed.

E.2. State of art

Firstly, it is necessary to have a background on how package managers work and what are the most important ones. A package manager is a collection of tools which automate software installation, uninstallation and update processes. That software is in repositories and package manager get them though metadata which contains package name, version and dependencies, which are the packages that a package needs to work (a package represents a collection of files which composes certain software). Once all the information is collected, package manager starts downloading content and installing packages as well as registering the operations performed.

One of the well-known package manager is *APT* (Advanced Packages Tool), which was created in the context of Debian project. One of its relevant features is that it does not treat packages individually and it treats them as a whole to get the best combination available. To make this work, it needs a source list file to indicate where repositories are, located in `/etc/apt/source.list`. Packages have a common structure, and from the content specified in sources file, it is easy to extract package metadata to process them.

One of the applications which use *APT* and the most common one is `apt-get`. It has lots of functionalities but the most commons are install, remove, update (to update the source list) and upgrade (to update packages). *APT* also allows package verification via digital signatures though `apt-key` which maintains public *GPG* keys used to verify signatures. *APT* also has user interfaces as aptitude (which uses the console in a semigraphic mode) and synaptic which is a graphic user interface (*GUI*).

Another package manager is *YUM* (Yellow Dog Updater) which is the package manager used in Linux systems based in *RPM*. Options provided are the same but its internal

behavior is different. *YUM* uses a configuration file in `/etc/yum.conf` which allows defining the global configuration and it is recommended to define different files for each repository. Then, when a package is asked to install, *YUM* firstly downloads the header where metadata is, to take decisions. In general, although metadata changes, options provided are more or less the same and this indicates the importance of having a good metadata definition to make the program.

Finally, *Microsoft* is trying to design its own package manager, called *OneGet* and it is intended to appear for the first time in *Windows 10*. *OneGet* will be part of Windows scripting language and commands shell, called *PowerShell*, which uses its scripts called cmdlets to perform the package manager operations. One of the advantages it will provide is that developers will be able to create their own repositories to make installation easier while *Microsoft* will maintain its own repository.

E.3. Metadata definition

The first step of the project is to define the metadata structure the program will use. Data format will be `.json` as it is suitable to interchange data and its dictionary structure can be easily parsed using *Python*, the programming language that will be used as it is also a scripting language as *TCL*, which was used to the first implementation of the package manager (called *myum*), but it provides more extensibility and it has more libraries and support to be able to give more functionalities.

The first file to define is the mirrors file which will be used to know in which repository each package in each version is. In this file, repositories will be specified and the program will be able to find packages inside them. Necessary information is the repository name and its path. Additionally, there is another field to enable/disable a repository. Paths to these repositories could be remote (they could have an *ssh* address to the server *VMGitLab* or a local address).

After defining how to get access to the packages, it is needed to set a global mechanism to define versions. *Myum* will use semantic versioning [13] , which uses three digits for each version (X.Y.Z) where the first one is changed when there is an important change which makes code incompatible, the second represent minor compatible changes and the third one is only changed when there are bug fixes.

Once having defined versions, tags must be defined in concordance with them to provide a clear way to download packages. Previously, `tag_name_vversion` (ex: `tag_pkg_v1.0`) format was used. As it is necessary to change format because versioning format was different before, with only two digits, now format will be `mtag_name_version` (ex: `mtag_pkg_1.0.0`).

As regards packages, it is required a metadata structure which defines the content to install. For each version a file `package.json` is defined with a field with its dependencies, the data that should be copied from the repository (including source, destination, type of content (file or directory) and a code to indicate the permissions the content needs) and the commands that must be executed when installing a package.

Whenever users want to install a package or a list of packages, he can define the installation in what it is called the install definition file. This file contains a list of packages and versions that are required to install but there is also the case when a user is developing a package and only wants the dependencies of the developing packages. In that case, version used is `d.d.d` to represent that only dependencies will be installed.

Finally, the last metadata file needed is the database which stores the installed packages and the operations performed. This file is very useful because, for instance, a package can only be uninstalled if it is already installed and there must be a file which stores that information. For this reason `.myum.db` file stores the installed packages including name, version, type (if the package was requested by the user or it was installed because it was a dependence of another requested package), installation files and the path of files installed (which are used to check package integrity). The history information also stores reinstall, uninstall or update operations and it is only used to report the user.

E.4. First functionalities of the program

Once metadata is defined, different options of *myum* can be implemented. Firstly, the command-line version will be explained and as the lower level functions are the same in graphic version, it will be easier to explain that version later.

As in every program, the first time user runs it, it is normal to use the help option to know about the program. Figure 51 shows the available options.

After the help option, the second displayed option is used to know *myum* version. It only displays a message in format `'myum version a.b.c'` (where a.b.c is the version in semantic versioning).

The following option (`-lp` or `--list_packages`) is used to know what packages can be installed and where they are. This option makes calls to `git ls-remote` command to retrieve the tags in the enabled repositories found in the mirrors file and as the tag name contains package name and version, it can be parsed to obtain the information needed to display a list with the package names, versions, repository names and repository addresses.

Options `-m` (or `--mirrors`) and `-d` (or `--definition`) are used to set the metadata files. By default, mirrors file is in a static path and definition file is `myum_install_def.json` file in the folder where *myum* was called. To make possible to define new mirrors file (for example if a user wants to use his local repositories), mirrors option can be used as well as the definition one to change the install definition file.

E.5. Package installation

The most important functionality which represents the program core is the one used to install a package. It consists on obtaining a list of files from a repository and downloading and outputting them in a specified path. This task is done in the following twelve steps:

```
$ python3 myum.py -h
usage: myum [-h] [-V] [-lp] [-m MIRRORS_FILE] [-i] [-d INSTALL_DEF_FILE]
            [-ck PACKAGE_TO_CHECK] [-f] [-u UNINSTALL_PACKAGE] [-li]
            [-s INSTALL_SPECIFIC_PKG] [-na] [-cl] [-nv] [-up] [-ar] [-hs]
            [-db] [-dr]

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -lp, --list_packages  List all the available repositories and packages
  -m MIRRORS_FILE, --mirrors MIRRORS_FILE
                        Change file containing the repositories mirrors list.
                        Default: /home/eda/myum/myum_mirror_list.json
  -i, --install         Install packets according to your install definition
                        file
  -d INSTALL_DEF_FILE, --definition INSTALL_DEF_FILE
                        use a specific install definition file instead of the
                        default (myum_install_def.json).
  -ck PACKAGE_TO_CHECK, --check PACKAGE_TO_CHECK
                        Check if the package installed has changed after
                        installation or if it remains the same.
  -f, --force           Force to install a package.
  -u UNINSTALL_PACKAGE, --uninstall UNINSTALL_PACKAGE
                        Uninstall a package.
  -li, --list_installed
                        Shows a list with the packages installed.
  -s INSTALL_SPECIFIC_PKG, --specific INSTALL_SPECIFIC_PKG
                        install a specified package given in the command line,
                        and ignore install definition file. Use format
                        pkg_a.b.c.
  -na, --noautomatic    not resolve dependencies automatically (myum will ask
                        you in case of conflicts).
  -cl, --clean          Clean cache directory.
  -nv, --noverbose      not print detail information of what is done.
  -up, --update         update all packages installed to its highest
                        compatible version.
  -ar, --autoremove     removes packages that were installed automatically to
                        satisfy dependencies from other packages and that now
                        they are not necessary
  -hs, --history        print history information contained in
                        myum_install_log (.myum.db)
  -db, --debug          enable debug mode
  -dr, --dry            simulate an myum operation without making changes
```

Figura 51: Help option of *myum*.

1. Check that a repository is clean (there are not changes to commit [1]). If it is not clean, installation can only continue with `-f` (or `--force`) flag.

2. Elaborate the collection of the available packages (this is done as if the list packages option was called, but without printing the results).

3. Obtain the list of packages that are going to be installed from the install definition file.

4. Obtain and solve the dependencies tree. This is the most challenging part of installation as there can be conflicts between incompatible versions. To do it, the list of all dependencies is obtained (dependencies tree) and then, when there is more than one version for the same package, if versions are compatible, the highest compatible is set as the final version. If they are incompatible, *myum* tries to find if the same conflict occurred before to solve in the same way as previously and if it is the first time, user will be asked as if no automatic mode is used where user will be always asked even with compatible versions.

5. Analyze if a package should be installed or not according to the installed information (if a package is already installed, it may not be installed again). Unless force option is enabled, a package will only be installed again if it is going to be installed in a higher compatible version.

6. Check package integrity (check if there is any new file or a installed file has been modified or deleted).

7. Uninstall installed package which are going to be installed in another version.

8. Download content from repositories and output them in the workspace directory.
9. Set reading, writing and execution permissions to installed files.
10. Execute required commands to complete installation (normally post-installation scripts).
11. Update `.myum.db` with the installed packages and the operations performed.
12. Update install definition files if proceed with the install currently configuration. At the end, this file will contain the name of installed requested packages with the version selected (not the original one).

E.6. Complementary functions of the program

After installing a package, there can be other possible operations to perform such as uninstall it, update it and so on. They are a very good complement to make myum a professional package manager.

One of these functionalities allows a package to be checked (`-ck` or `--check`) if its files have been modified or deleted or if there are new files. To do that, files information contained in `.myum.db` is retrieved and *MD5* hash is calculated for every installed file. Then, these *MD5* hashes are compared to the hashes obtained in the remote files. If all hashes matches and there are not deleted files nor new, integrity check is right.

Another important functionality is to uninstall a package (`-u` or `--uninstall`). This process is simpler than installation and only consists of four steps. The first one is to check package integrity to avoid deleting modified or new files. If everything is right, package information is obtained in the `package.json` file and files are directories are deleted. Finally, `.myum.db` is updated to register uninstall operations.

Another option allows the user to see the installed packages (`-li` or `--list_installed`) in a list which provides name, version, type of package (requested or dependence) and installation date and time. Furthermore, `-hs` or `--history` allows to see a list of the registered operations in which appears operation type, package name, version, type of package and date and time.

There is also the possibility to install a specific package without using the install definition file with the specific option (`-s` or `--specific`), which allows indicating a package by command line in `package_a.b.c` format.

When installation is performed, there are lots of files that are downloaded to a cache directory that are not used after the installation. Clean option (`-cl` or `-clean`) allows deleting the whole content of that directory.

If a package is installed but there is a higher compatible version, there is also an option to update it (`-up` or `--update`). This option gets the highest compatible version to each installed version of each package and if there is anything new and compatible, it updates

it. The process is the same as installation because myum generates a list with the packages that can be updated and treat them as if they were the content of the install definition file.

In the uninstallation process described before, when a package was deleted, its dependencies were maintained because they could be needed by other packages. Autoremove option (`-ar` or `--autoremove`), checks every package with `'dependence'` type and if that is the case that there is not any requested package which depends on that package at any level in the dependencies tree, then that package is removed. Developing packages (which have `d.d.d` as version) are also taken into account to avoid removing their dependencies.

Finally, to complement operations, `-f` (or `--force`) option allows to force an installation, which means to continue the process even if the repository is not clean, integrity check fails or version required is older or not compatible with the version installed. Furthermore, `-na` or (`--noautomatic`) allows the user to be asked to resolve any conflict when solving dependencies independently of the possible versions, which could be compatible or not (by default, user is only asked if versions are compatible and there is not a previous evidence of what to do).

Myum also allows the user to define what amount of information is desired to be displayed when executing a command. Verbosity levels are warning (which only shows warnings and errors), info (which display information messages and the what warning level shows) and debug which shows apart from what info level shows, the output produced of external commands that *myum* execute at the end of installation.

The last option (`-dr` or `--dry`) allows the user to simulate an operation. If this option is used in an installation, all installation processes will be performed but no data will be copied and *myum* will not register any operation. Therefore, this option simulates and displays the results of the operations needed for the different operations but do not make any change (it does not install, update, uninstall or reinstall any package).

E.7. Graphic interface

After implementing the command-line version, graphic interface can be developed. `Tkinter` library of *Python* is used to do this task. Figure 52 shows a general view of the window displayed. As it can be easily seen, there are different areas:

- Menu bar: It has three different menus: *File*, *Tools* and *Help*. *File* menu allows changing the mirrors file, the install definition file or the workspace. *Tools* menu includes basic operations as list packages, performs operations (called 'apply' in this mode), update packages, autoremove or show history. *Help* menu shows program help and version.
- Options bar: This bar allows executing important operations of the program as the *Tools* menu, but also includes options to define verbosity or enable/disable force, no automatic and dry flags.
- Main panel: This panel shows alphabetically all the available packages and what packages are installed. It allows checking package integrity in installed packages and, what it is the

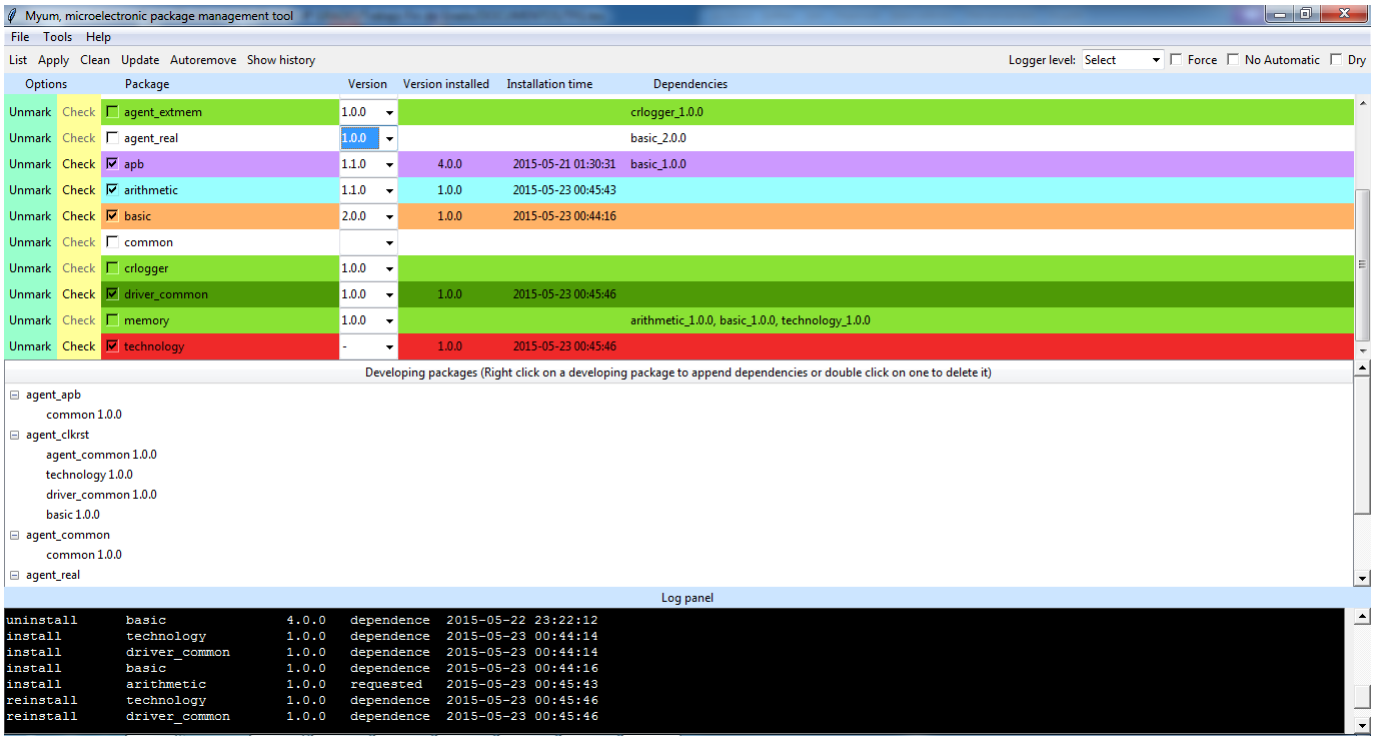

Figura 52: Graphic user interface.

most important feature, it allows marking packages to do certain operations. In version column, user can select one version among the available ones. When this happens, package is marked to be installed in that version. If it is not installed, it will be marked to install in green, but if it is already installed, it will be marked to reinstall if it is installed in the same version or to update if version is different. In this case, marks have different colors depending on if version selected is newer, older, compatible or not with the installed version. There is also a `'-'` alternative in versions list to mark a package to remove. After packages have been marked, user can select *Apply* option to make all changes. Developing packages with `d.d.d` are not marked directly in this panel but they are taken into account when using 'apply' option. Figure 52 shows different package marks.

- Developing packages window: Shows what packages are being developed (packages which have a `package.json` in `tools/myum` directory) and their dependencies. This window allows appending or deleting dependencies to those packages without needing to edit `package.json` file.

- Messages panel: Shows messages while performing operations in *myum* as it was the console mode (report messages are the same). It can also interact with users when there is any conflict between versions when solving the dependencies tree or in case a package is marked to install and uninstall at the same time (it could happen that a package is marked to uninstall but it is also a dependence of a package marked to install).

E.8. Complementary project development and results

Code to implement *myum* functionality is not the only task to do in the project. There are also other parts to implement to be able to maintain the project in the future. The first of them is the unit tests.

Unit tests are implemented code which allows testing small parts of the source code. In this project, a unit test has been implemented for every function. Some tests cover a small part of code, while others, as they test functions which integrate others, have a higher coverage. To measure total coverage (line of tested lined / total lines of code), *coverage* tool has been used, which can generate *HTML* files reporting which lines are tested and percentages. Results show a 98 % of coverage, which is acceptable since it is impossible to reach 100 % with the program because if measure is carried out in *Linux*, it is impossible to test *Windows* lines and vice versa.

Another important aspect that was taken into account was the compatibility issues. Compatibility between *Python* versions was easy to handle since changes were only in library names, but when talking to different operative systems, situation was different. Most important problems were that paths are composed differently in *Windows*, which use backlash, and *Linux* or *Cygwin* which user `'/'` bar. Another compatibility problem was that paths were also different (`T:\project_X` in *Windows*, was `/cygdrive/t/project_X` in *Cygwin* and `/home/projects/project_X` in *Linux*). End of lines and permissions systems are also different. Scape sequences used to display messages in different colors are not available in *Windows* and external commands could also vary between platforms. Therefore, there has been needed to be very careful to all these issues.

Finally, it is necessary to make reference to the code documentation. It is something done while implementing the code, not at the end as it is mentioned in the document, but its importance now is the fact that there is a standard way of documenting in the section and comments have been adapted to *Doxygen* format to make all documentation from different projects uniform.

At the end of the project, there are two main files `myum.py` and `myum_gui.py` which can be executed and run command-line version and graphic one, respectively, which are consistent and provide the functionalities mentioned in the previous sections.

Users have started using *myum* and have showed their approval after installing packages in their projects. At first, they proposed improvements as changing text messages that were not clear or changing the warning color (which was originally yellow and was not seen correctly in terminals with white background color). Furthermore, they were who proposed the inclusion of the developing packages window in graphic mode since previously graphic mode only performed operations but dependencies needed to be appended or deleted modifying files manually. What is more, they could find some unexpected bugs that were fixed to end up with a robust program which satisfies initial necessities exposed at the beginning and contributes to the improvement in the section.

E.9. Conclusions

The main purpose of this Bachelor Thesis has been to design a microelectronic tool which handles *IP* packages and allows user performing operations as install, uninstall... To achieve this goal, *myum* has been implemented in two separated but consistent versions: command-line and graphic one. In order to implement it, firstly a metadata structure was designed. Afterwards, main functionalities were implemented to list packages, install and define metadata files and finally they were complemented with options to uninstall, check integrity and so on. To conclude, unit tests were implemented to check the program and to be able to maintain it in the future.

The most challenging part of the project has been to familiarize with the project organization in *Crisa*, *Git* and with *Python* libraries to know what it is needed to do. Once knowing how to face those problems, *myum* could be finished and now it is being used by different workers in their projects and they consider this work has a lot of benefits when they prepare their projects. Therefore, if objectives of this Bachelor Thesis are taken into consideration, it is certain that in general they are accomplished and day-by-day experience will measure better how effective *myum* is and if there is anything to change in the future or not. As far as it can be said, it has been worth doing the project since apart from the academic and professional benefits at a personal level, the program is going to be useful and other people are going to use it. Finally, the project could be complemented with another tool to configure a project at the beginning which manages the available applications, where myum would be one of them.

Referencias

[1] Scott Chacon. *Pro Git*. Git, 2009.

[2] OpenSUSE community. Package management, November 2013. URL: https://en.opensuse.org/Package_management.

[3] Members of Ubuntu Documentation Project et al. *Ubuntu Server Guide*. Ubuntu, 2014.

[4] Eric Foster-Johnson et al. *RPM Guide*. Fedora, 2011.

[5] Raphaël Hertzog et Roland Mas. *The Debian Administrator's Handbook*. Debian, 2013.

[6] Gustavo Noronha Silva et Hugo Mora. *APT Howto*. Debian, 2003.

[7] Ellen Siever et al. *Linux in a nutshell*. O'Reilly, 2009.

[8] Barbora Ančincová et al. *Red Hat Enterprise Linux 6 Deployment Guide*. Red Hat, 2015.

[9] Balakrishnan Mariyappan. 15 Linux Yum Command Examples – Install, Uninstall, Update Packages. The Geek Stuff, August 2011. URL: http://www.thegeekstuff.com/2011/08/yum-command-examples/.

[10] Microsoft. Scripting con Windows PowerShell, August 2014. URL: https://technet.microsoft.com/es-es/library/bb978526.aspx.

[11] Chris Hoffman. Windows 10 Includes a Linux-Style Package Manager Named "OneGet", October 2014. URL: http://www.howtogeek.com/200334/windows-10-includes-a-linux-style-package-manager-named-oneget/.

[12] *Cygwin User's Guide*. Red Hat, 2015.

[13] Creative Commons CC BY 3.0. Semantic Versioning 2.0.0, 2014. URL: http://semver.org/.

[14] Fred L. Drake et al. Python 3.4.3 documentation, 2015. URL: https://docs.python.org/3.4/index.html.

[15] Fred L. Drake et al. Python 2.7.10rc0 documentation, 2015. URL: https://docs.python.org/2.7/index.html.

[16] GNU Free Documentation License 1.3 or late. Color Bash Prompt, 2015. URL: https://wiki.archlinux.org/index.php/Color_Bash_Prompt.

[17] Daniel Burrows. Modelling and Resolving Software Dependencies. 2005.

[18] William E. Shotts Jr. *The Linux Command Line*. Creative Commons, 2013.

[19] Yong-Xia Z. Ge Z. et Ge Z Yong-Xia, Z. Md5 research. In *Second International Conference on Multimedia and Information Technology. IEEE*.

[20] National Institute of Standards and Technology. Secure Hashing, 2012. URL: http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[21] Dustin Hurlbut. *Thumbs db files forensic issues.* Access Data Training, 2005. URL: https://ad-pdf.s3.amazonaws.com/wp.Thumbs_DB_Files.en_us.pdf.

[22] Tutorialspoint. Python GUI Programming (Tkinter). URL: http://www.tutorialspoint.com/python/python_gui_programming.htm.

[23] Ned Batchelder. *coverage.py Documentation.* Read the Docs, 2015.

[24] Dimitri van Heesch. Doxygen, 2014. URL: http://www.stack.nl/~dimitri/doxygen/.

[25] Project Management Institute. *Guía de los Fundamentos para la Dirección de Proyectos (Guía del PMBOK).* 2009.

[26] Álvaro Cuervo García. *Introducción a la administración de empresas.* Civitas, 2008.

[27] JetBrains Community. Quick Start Guide. Exploring the IDE, 2013. URL: https://www.jetbrains.com/pycharm/quickstart/.

[28] Grupo de Políticas Públicas y Regulación. *La gestión de derechos de propiedad intelectual en el entorno TIC.* Colegio Oficial de Ingenieros de Telecomunicación, 2014.

[29] Jefatura del Estado. *Vigente Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal.* Boletín Oficial del Estado, 1999.

[30] Grupo de Políticas Públicas y Regulación. *Protección de datos y privacidad en el sector TIC.* Colegio Oficial de Ingenieros de Telecomunicación, 2014.

[31] Computadoras Redes e Ingeniería. Descripción de la empresa. URL: http://www.crisa.es/homeE.htm.